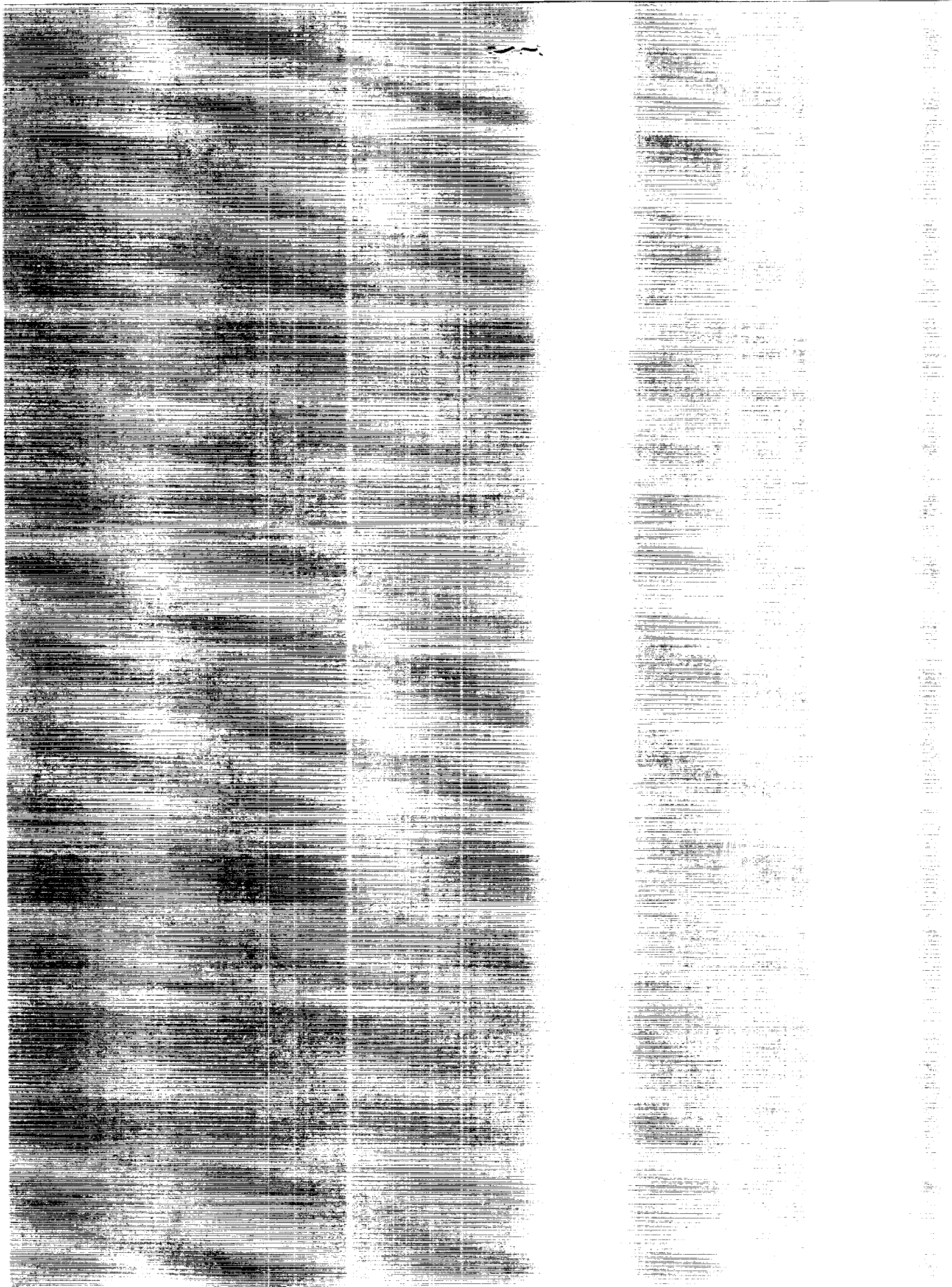
NASA Cont[illegible]

Accuracy [illegible] the Cheb[illegible]

Wai Sun Don [illegible]

GRANT NAG1-[illegible]
DECEMBER 1991



NASA Contractor Report 4411

Accuracy and Speed in Computing the Chebyshev Collocation Derivative

Wai Sun Don and Alex Solomonoff
Brown University
Division of Applied Mathematics
Providence, Rhode Island

Prepared for
Langley Research Center
under Grant NAG1-1145

National Aeronautics and
Space Administration

Office of Management

Scientific and Technical
Information Program

1991

1 Introduction

A problem frequently encountered by scientists and engineers is the solution of partial differential equations. Since these equations often don't have closed form solutions, they are often forced to compute approximations to the solution numerically. One category of method for approximating the solution of PDE's is spectral methods, which are described in many texts, for example (Gottlieb and Orszag, 1977). In these schemes, the solution is assumed to be a finite linear combination of some set of basis functions, and the PDE is expressed in terms of the coefficients. This gives a set of equations for the coefficients which is solved by other appropriate numerical method relevant to the problem.

These schemes can be very efficient because the rate of convergence (or the order of accuracy) depends only on the smoothness of the solution as the number of grid points or modes increases. This is known in literature as *spectral accuracy*. If the solution is smooth, (and the basis functions are too) only a few basis functions are needed to accurately approximate the solution. In particularly, if the solution of the PDE is analytic, the error decays exponentially. In contrast to finite difference method, the order of accuracy is fixed by the scheme.

A slightly different class of methods is collocation methods. Instead of working in the coefficient space, the function is discretized in the physical space at some chosen set of collocation points, for example, Chebyshev or Legendre points. The solution is forced to satisfy the PDE only at the collocation points. This methods has the advantage of being able to deal with nonlinear terms and/or general forcing function more easily than the methods describe earlier while retaining spectral accuracy. Readers interested in this subject are referred to (Canuto, et. al. 1988, Gottlieb and Orszag, 1977) for detail.

In this paper, we study issues that arise from Chebyshev collocation methods. An important operation in these methods is to compute the derivative of a function at the collocation points. It is a surprisingly complex subject. There are three major ways of computing it, and several variations within each type. All told, there are over a dozen different algorithms possible. They differ considerably in the amount of roundoff error they produce, the amount of memory they use, and the amount of CPU time they require. The two algorithms generally used in practice are matrix-vector multiplication (MV) and transform-recursion (FFT) algorithm. These two algorithms will be described in detail later.

The roundoff error is an important issue. The Chebyshev derivative is a rather ill-conditioned operator, and inaccuracies in the function can be magnified by as much as $O(N^4)$, where N is the number of collocation points, when computing the derivative using the matrix-vector multiplication method. As more powerful computers become available, calculations using more and more interpolation points are attempted for solving complicated physical phenomenon, and if N is as

large as 100 or 1000, rounding errors in the computation can cause serious losses of accuracy.

Past research has addressed the problem of roundoff error in Chebyshev collocation methods and various algorithms were proposed in attempted to alleviate problem associated with the roundoff error (Breuer and Everson, 1989, Rothman E.). The best result managed to reduce the roundoff error from $O(N^4\epsilon)$ to $O(N^3\epsilon)$, where ϵ is machine precision, for the matrix-vector multiplication algorithm. Hence, only the transform-recursion algorithm, which have roundoff error of $O(N^2\epsilon)$, was recommended for computing Chebyshev spectral derivative regardless of the efficiency of the algorithm on different computing platforms.

However, our own investigation of the roundoff error leads us to discover a subtle numerical error associated with in computing the elements of the full differentiating matrix. We show that this numerical error is the source of the $O(N^3\epsilon)$ roundoff error. By flipping the upper half of the matrix to replace the lower half, our numerical results showed that the roundoff error of matrix-vector multiplication is at least as small as the transform-recursion algorithm. This in turn showed the roundoff error of the matrix-vector multiplication is only $O(N^2\epsilon)$.

We also derived an estimate of the minimum possible roundoff error in a statistical sense. Our new algorithm and transform-recursion algorithm agreed with this estimation quite well. This implies these algorithms are doing the best one can ask for. Any further reduction in rounding errors must come not by any modification of the algorithm but the modification of the Chebyshev collocation method itself.

Speed is also an important issue. Using the naive matrix-vector multiplication algorithm, computing the Chebyshev derivative takes $O(N^2)$ time. In many situations this would be a major component of the CPU time used, and so minimizing it would be important. The algorithms that use cosine transforms take $O(N \log N)$ time, but for the values of N that people actually use, they are not necessarily the fastest algorithms. This is especially true when the computation is done on a vector computer, since the cosine transform does not vectorize as easily as the matrix multiplication. To obtain a reasonable efficiency in computing solution of partial differential equations, one must aware of the interplay between accuracy desired and the CPU time usage on a particular computing platform.

This paper is organized in the following sections: Section 2 will introduce Chebyshev collocation method and algorithms in computing the Chebyshev spectral derivative. Section 3 describes the preconditioning method to reduce the roundoff error. The statistical minimum possible roundoff error is derived in section 4. In section 5, we will discuss the numerical results of the accuracy using different algorithms. We also will describe the numerical error associated with in computing the elements of the differentiation matrix and ways to overcome it. Section 5 will give the final conclusion for the study of the roundoff error. Conclusion about accuracy is given in section 6. The CPU timing of various algorithms and grid sizes on IBM 9121-320 VF and Cray 2 are discussed in

section 7. Section 8 presents the conclusion of CPU timing.

2 Chebyshev Collocation Methods and Differentiation

In this section, we will present the Chebyshev collocation method. Given a smooth function $v(x)$ in the domain $x \in [-1, 1]$, we shall consider the Chebyshev-Guass-Lobatto collocation points,

$$x_i = \cos(\frac{\pi i}{N}), \qquad i = 0, \ldots, N\,.$$

They are the extrema of the N order Chebyshev polynomial

$$T_N(x) = \cos(N \cos^{-1} x)\,.$$

The function $v(x)$ is interpolated by constructing the N order interpolation polynomial $g_j(x)$ such that $g_j(x_k) = \delta_{jk}$, i.e.

$$u(x) = \sum_{j=0}^{N} u_j g_j(x) \tag{1}$$

where $u(x)$ is the polynomial of degree N and $u_j = v(x_j)$, $j = 0, \ldots, N$. It can be shown that

$$g_j(x) = \frac{(-1)^{j+1}(1-x^2)T'_N(x)}{c_j N^2 (x - x_j)}, \qquad j = 0, \ldots, N\,, \tag{2}$$

where

$$c_j = \begin{cases} 2 & j = 0, N \\ 1 & j = 1, \ldots, N-1 \end{cases}\,.$$

The derivative of $u(x)$ at the collocation points x_j's can be computed in many different ways. However, we will only present three algorithms and describe them in some detail. They are the Matrix-Vector Multiplication Method (MV), the Even-Odd Decomposition Method (EO) and the Transform-Recursion Method (FFT). Algorithms other than these have been considered, but don't appear to offer any advantages over the ones presented here (Breuer and Everson, 1989).

Matrix-Vector Multiplication Method

The most obvious way to compute the derivative is the matrix-vector multiplication. The entries of the Chebyshev derivative matrix D, D_{jk}'s are computed by taking the analytical derivative of $g_j(x)$ and evaluate it at collocation point x_k for $j, k = 0, \ldots, N$, i.e., $D_{jk} = g'_j(x_k)$. Then the entries of the matrix are

$$\begin{aligned} D_{jk} &= \frac{c_j}{c_k}\frac{(-1)^{j+k}}{(x_j - x_k)} && j \neq k \\ D_{jj} &= \frac{-x_j}{2(1-x_j^2)} && j \neq 0, N \\ D_{00} &= -D_{NN} = \frac{2N^2+1}{6} \end{aligned}\,, \tag{3}$$

and now the derivative of $u(x_i)$ becomes

$$u_i' = \sum_{j=0}^{N} D_{ij} u_j, \qquad i = 0, \ldots, N. \tag{4}$$

Even-Odd Decomposition Method

Another algorithm is the even-odd decomposition algorithm, as discussed in (Solomonoff, 1992). It exploits the following regularity that the Chebyshev derivative matrix has, namely

$$D_{ij} = -D_{N-i,N-j}.$$

Actually, this is a property that most derivative matrices have, not just the Chebyshev. The following description is for an even number of interpolation points, i.e., N is odd. If the number of points is odd, the algorithm is slightly more complicated, and the reader is referred to (Solomonoff, 1992). The algorithm has three stages. First, the vector u is decomposed into its even and odd parts:

$$\begin{aligned} e_i &= u_i + u_{N-i} \\ o_i &= u_i - u_{N-i} \end{aligned} \quad , \qquad i = 0, \ldots, \frac{N-1}{2}.$$

Next, e and o are multiplied by matrices related to the original derivative matrix:

$$\begin{aligned} e' &= D_e e \\ o' &= D_o o, \end{aligned}$$

where

$$\begin{aligned} (D_e)_{ij} &= \tfrac{1}{2}(D_{ij} + D_{i,N-j}) \\ (D_o)_{ij} &= \tfrac{1}{2}(D_{ij} - D_{i,N-j}) \end{aligned} \quad , \qquad 0 \le i,j \le \frac{N-1}{2}.$$

Finally, u' is constructed from e' and o':

$$\begin{aligned} u_i' &= o_i' + e_i' \\ u_{N-i}' &= o_i' - e_i' \end{aligned} \quad , \qquad i = 0, \ldots, \frac{N-1}{2}. \tag{5}$$

This algorithm has the advantage of only using half as many operations and half as much memory as the matrix-vector multiplication algorithm.

Transform-Recursion Method

There is an algorithm involving fast Fourier transforms. The polynomial interpolating the points is expressed as a sum of Chebyshev polynomials

$$u_i = \sum_{k=0}^{N} \tilde{u}_k T_k(x_i), \tag{6}$$

and the coefficients can be computed by

$$\tilde{u}_j = \frac{1}{c_j N} \sum_{k=0}^{N} \frac{1}{c_k} u_k T_j(x_k) .$$

Since

$$\begin{aligned} T_j(x) &= \cos(j \cos^{-1} x) , \\ T_j(x_k) &= \cos(\frac{\pi j k}{N}) , \end{aligned}$$

and we have

$$\tilde{u}_j = \frac{1}{c_j N} \sum_{k=0}^{N} \frac{1}{c_k} u_k \cos(\frac{\pi j k}{N}) . \tag{7}$$

Except for the c_k, this is a cosine transform, which can be evaluated in $O(N \log N)$ operations using fast Fourier transforms.

Next, the coefficients of the derivative of the interpolating polynomial are obtained by a recurrence relation:

$$\begin{aligned} \tilde{u}'_N &= 0 , \\ \tilde{u}'_{N-1} &= N \tilde{u}_N , \\ c_k \tilde{u}'_k &= \tilde{u}'_{k+2} + 2(k+1)\tilde{u}_{k+1} , \qquad k = N-2, \ldots, 0 \quad . \end{aligned} \tag{8}$$

Finally, u' is obtained by an inverse cosine transform:

$$u'_i = \sum_{j=0}^{N} \tilde{u}'_j \cos(\frac{\pi i j}{N}) . \tag{9}$$

This algorithm takes $O(N \log N)$ operations, compared with $O(N^2)$ operations for the matrix-multiply algorithms. How fast it actually is depends mostly on the quality of your FFT software.

All of these algorithms have rather poor performance with respect to roundoff error. Considering that the spectral radius of D is $O(N^2)$ (Trefethen and Trummer, 1987), or alternatively, that the largest elements of the matrix are $O(N^2)$, one would expect that the roundoff error would grow approximately like $N^2\epsilon$, where ϵ is the machine precision. (The machine precision is the smallest positive number such that $1 + \epsilon > 1$ in the arithmetic of the computer.)

(Breuer and Everson, 1989) found that for a few points near the edges of the domain, the error was $O(N^2\epsilon)$, and for the others in the interior of the domain, the error was more like $O(N\epsilon)$. This was for the transform-recursion algorithm. The error for the matrix multiply was found to be approximately $O(N^4\epsilon)$. In order to reduce the error, they considered several variations on the

transform-recursion algorithm. This included several modified versions of the recursion, and an algorithm consisting of a cosine transform, $O(N)$ operations on the transformed data, an inverse sine transform, and then $O(N)$ operations on the inverse transformed data. The speed and accuracy of all these variants was about the same as the unmodified transform-recursion algorithm.

3 Preconditioning

(Rothman) has considered another approach to reducing the roundoff error. He noticed that since the entries of the derivative matrix near the upper left and lower right corners are large, modifying the process to reduce their influence might reduce the roundoff. The following technique is a modified version of that described in (Rothman).

The function to be differentiated is modified by subtracting off a linear function which interpolates $u(x)$ at the endpoints:

$$h(x) = u(x) - \frac{1}{2}(u(1) + u(-1)) - \frac{x}{2}(u(1) - u(-1)).$$

Then $h(x)$ is differentiated in the usual way, and the derivative of the linear function is added on.

$$u'(x) = h'(x) + \frac{1}{2}(u(1) - u(-1)).$$

If $h = \{h(x_0), \ldots, h(x_N)\}$ and g is a vector whose elements are all ones, then the discrete version of this is

$$u' = Dh + \frac{1}{2}(u(1) - u(-1))g. \tag{10}$$

$h(\pm 1) = 0$ and small near ± 1. This means that the large elements of the derivative matrix near the corners of the matrix are multiplied by small numbers when the matrix-vector multiplication is carried out. Hopefully this means that the influence of these elements is reduced.

4 Minimum Possible Roundoff Error

It is impossible to judge the accuracy of any of these algorithms unless we know what the best possible results are. By taking a statistical approach, it is possible to estimate the minimum possible roundoff error that an algorithm for computing the Chebyshev derivative could have. If an algorithm achieves it, then we know that there is little hope of modifying it to reduce the error. Or, to say it in a more constructive way, the only way to reduce the roundoff error would be to change the Chebyshev derivative itself, rather than just the algorithm for computing it.

Suppose that u is a vector of function values in infinite precision. And suppose that $\bar{u}$ is the same vector of function values, but rounded to finite precision. They differ by a vector $e = u - \bar{u}$. If $\bar{u}$ is used to compute the Chebyshev derivative of u, then there will be an error $D(u - \bar{u}) = De$.

The error in an actual computation would be at least as big as this, since in addition to the error associated with the finite precision of u, there is the error from the finite precision of D and error in summing the inner products in the matrix-vector multiplication. So how big is De?

We can answer this by thinking of e as a random vector. The elements of e will be assumed to be uncorrelated, zero-mean random variables with variance equal to ϵ, the machine precision. So if C is the correlation matrix of e, then $C = E(ee^T) = \epsilon^2 I$, where $E(\cdot)$ is expectation.

If e is a random vector, then so is De. What is its mean and correlation matrix? In general, if a random vector x has mean μ and covariance matrix C, then Ax has mean $A\mu$ and covariance matrix ACA^T. So the mean of De is zero, and its covariance matrix is $\epsilon^2 DD^T$.

If C is the correlation matrix of x, then C_{ii} is the expectation of the square of the $i-$th element of x. This is a measure of how big x_i is, on the average. So a reasonable estimate of the maximum norm of the roundoff error of Du is

$$\sup_i \epsilon \sqrt{(DD^T)_{ii}}\,, \tag{11}$$

and a reasonable estimate of the L_2-norm of the roundoff error is

$$\epsilon \left[\sum_{i=0}^{N} (DD^T)_{ii} \right]^{\frac{1}{2}}. \tag{12}$$

This can be evaluated, at least numerically, and compared to the roundoff error of the different algorithms.

5 Results About Accuracy

We have computed the Chebyshev spectral derivative for several different values of N using several algorithms and compared their accuracy. Two test function are used for this purpose. They are

$$u(x) = \sin(2x) + \cos(2x)\,, \tag{13}$$

$$u(x) = \exp(-x^2)\,. \tag{14}$$

Moreover, these computations are done on two different machines. The first is an IBM 9121-320 VF with a vector facility. This is Brown University's mainframe computer. The other machine is a Cray-2. This is Voyager at NASA-Langley Research Center. The computations were done in double precision on the IBM and single precision on the Cray. The machine precision for double precision on the IBM is $\epsilon = 2.1 \times 10^{-16}$ and single precision on the Cray is $\epsilon = 3.5 \times 10^{-15}$.

On the IBM, the three algorithms used were the matrix multiply, the even-odd, and the transform-recursion where the cosine transform was computed by symmetrically extending the input data and using a real-to-complex FFT algorithm. The matrix multiplications and the FFTs

were computed using ESSL (Version 4) subroutines. (ESSL is IBM's optimized scientific computing subroutine library.)

On the Cray, the same three algorithms were used. In addition, the transform-recursion algorithm was also computed using the forward and inverse cosine transform (CFT) subroutines FCR and FCRINV from LARCLIB. This is a library local to NASA Langley. They appear to do pre- and post-processing of the input data, and then call an FFT subroutine.

All of the algorithms were computed twice. Once with no preconditioning, and once using the preconditioning idea described in section 3. Two norms for the error were computed, the L_∞ norm, and the L_2 norm. The L_2 norm we used was not exactly the usual one. We used

$$\sqrt{\frac{\pi}{N}\sum_{i=0}^{N}\frac{1}{c_k}f_i^2}\,.$$

This is supposed to approximate the Chebyshev-weighted L_2 norm

$$\sqrt{\int_{-1}^{1}\frac{(f(x))^2}{\sqrt{1-x^2}}dx}\,.$$

The minimum error estimate described in section 4 was also computed and displayed on the same graphs with the other data. The estimated L_2 error was computed using the discrete Chebyshev L_2 norm above, rather than as in section 4. The dotted line represents the estimated minimum error. The curves with solid symbols represent algorithms with no preconditioning. The curves with hollow symbols represent algorithms with the preconditioning described in section 3.

Figures 1 and 2 present roundoff error as a function of N for all of the above cases. L_∞ error is computed for $u(x) = \sin(2x)+\cos(2x)$ on the IBM 9121-320 VF (figure 1) and $u(x) = \exp(-x^2)$ on the Cray 2 (figure 2). We computed error for several other combinations of error norm, machine, and function, but they all looked basically the same, so we are only presenting these two.

The worst accuracy is from the matrix multiply (MV) and even-odd (EO) algorithms with no preconditioning. Much better is the accuracy of the cosine transform-recursion (CFT) algorithm, with or without preconditioning. Best is the accuracy of the FFT algorithm, with or without preconditioning, and the two matrix algorithms (MV & EO) with preconditioning. The last four have about the same accuracy. The accuracy of the best algorithms is about the same or slightly worse than the estimated minimum error.

Use of Trigonometric Identities

Why is the error of the unpreconditioned matrix algorithms so bad? (Breuer and Everson, 1989) investigated this and concluded that the reason was that there were large errors involved in computing the elements of the matrix D. Computing the matrix elements involves computing

$$\frac{1}{x_i - x_j}\,.$$

This is hard to compute accurately. Since the smallest distance between points is $O(N^{-2})$, this involves taking a difference between two very similar numbers, and then taking the reciprocal amplifies the error by a large factor:

$$\begin{aligned}\frac{1}{x_i - x_j} &= \frac{1}{O(N^{-2}) + \epsilon} \\ &= \frac{O(N^2)}{1 + O(N^2\epsilon)} \\ &= O(N^2)(1 - O(N^2\epsilon)) \\ &= O(N^2) + O(N^4\epsilon) \quad .\end{aligned}$$

So the error in the matrix elements is $O(N^4\epsilon)$. This is rather large. Fortunately there is a way to compute the matrix elements more accurately.

Since the interpolation points are computed by a cosine function,

$$x_i = \cos(\frac{\pi i}{N}),$$

it is possible to use trigonometric identities to eliminate the subtraction of similar numbers (Canuto, et. al., 1988, pages 504,511,512, Rothman) :

$$\begin{aligned}x_i - x_j &= 2\sin\frac{\pi}{2N}(i+j)\sin\frac{\pi}{2N}(-i+j) \\ 1 - x_i^2 &= \sin^2(\frac{\pi}{N}k).\end{aligned} \tag{15}$$

Instead of (3), we have

$$\begin{aligned}D_{jk} &= \frac{1}{2}\frac{c_j}{c_k}\frac{(-1)^{j+k}}{\sin\frac{\pi}{2N}(j+k)\sin\frac{\pi}{2N}(-j+k)} \quad j \neq k, \\ D_{jj} &= -\frac{1}{2}\frac{x_j}{\sin^2(\frac{\pi}{N}j)} \quad j \neq 0, N, \\ D_{00} &= -D_{NN} = \frac{2N^2+1}{6} \quad .\end{aligned} \tag{16}$$

This formula should result in more accurate entries of the matrix. In (Breuer and Everson, 1989), the matrix elements were not computed with trigonometric identities.

Figures 3 and 4 are the same as the first two except that the matrix multiply and even-odd algorithms use matrices that have been calculated using this trigonometric identity form. The transform-recursion algorithms have been computed exactly as before. The accuracy of the two matrix algorithms (MV & EO) with preconditioning remains as good as the FFT algorithm. The accuracy of the matrix multiply algorithm (MV) without preconditioning improves considerably, but it is still the least accurate of all the algorithms. By contrast, the even-odd algorithm (EO) without preconditioning improves dramatically, becoming as good as the FFT algorithms and the preconditioned algorithms. This is true even though the matrices used in the even-odd algorithm were calculated from the matrix used in the matrix multiply algorithm. Why is there a difference?

Flipping of the Differentiation Matrix

A clue to the answer came from looking at the roundoff error at each point for the matrix multiply algorithm. We found that the error near $x = -1$ was larger than the error near $x = +1$, even if the function being differentiated was symmetric. Even with the trigonometric identity form of the derivative matrix, some elements of the matrix were still not being calculated accurately.

The problem lay in the fact that if x is a small number, then $\sin(x)$ and $\sin(\pi - x)$ are both (the same) small number. But, while $\sin(x)$ can be calculated with *relative* error comparable to machine precision ϵ, $\sin(\pi - x)$ can only be calculated with *absolute* error comparable to ϵ. Its relative error is $O(\epsilon/x)$.

In figure 5, the relative error in computing $\sin^{-2}(x)$ and $\sin^{-2}(\pi - x)$ is graphed as a function of x. The error for $\sin^{-2}(x)$ is roughly machine precision and doesn't depend on x. On the other hand, the error of $\sin^{-2}(\pi - x)$ grows as x gets smaller and the slope of the graph indicates that the error is roughly proportional to x^{-1}. Since this is the relative error, and $\sin^{-2}(\pi - x) = O(x^{-2})$, the absolute error is proportional to x^{-3}.

If i and j are small, then calculating D_{ij} involves calculating $(\sin(\delta_1)\sin(\delta_2))^{-1}$, where δ_1 and δ_2 are small numbers. This can be done accurately. But if i and j are both near N, then D_{ij} requires calculating something like $(\sin(\pi - \delta_1)\sin(\pi - \delta_2))^{-1}$. This cannot be done accurately. This results in a roundoff error of $O(N^3\epsilon)$ for the matrix elements in the lower right corner of D:

$$\begin{aligned} \frac{1}{(\sin(O(N^{-1})) + O(\epsilon))^2} &= \frac{O(N^2)}{(1 + O(N\epsilon))^2} \\ &= O(N^2)(1 - 2O(N\epsilon)) \\ &= O(N^2) + O(N^3\epsilon). \end{aligned}$$

The reason the even-odd algorithm (EO) had such good accuracy compared to the matrix multiply algorithm (MV) was that the matrices used in the even-odd algorithm were calculated using only the top half of the derivative matrix in (16), and didn't use the inaccurate lower half.

A simple remedy for this problem is not to calculate the bottom half of the matrix at all. Just calculate the top half, flip it over, and use it in place of the bottom half. That is, use the property

$$D_{ij} = -D_{N-i,N-j}, \qquad i = \frac{N}{2} + 1, \ldots, N$$

to replace the bottom half with the top.

Figures 6-9 show roundoff error for the different algorithms when the derivative matrices were calculated using both the trigonometric identity form and this flipping technique. We consider this case more important than the previous ones, so we have included more combinations of norm, machine and function.

Now the accuracy of the matrix multiply algorithm (MV) without preconditioning is better than that of the cosine transform algorithm (CFT), and as good as all the other algorithms. The

preconditioned algorithms offer no significant improvement in accuracy over the unpreconditioned ones. The accuracy of all of the algorithms, except for the cosine transform algorithm (CFT) is almost as good as the estimated lower limit.

Although the exact numbers are slightly different, all of the results in this section are true for both the IBM and the Cray computer.

6 Conclusions About Accuracy

The preconditioning technique is able to restore good accuracy when applied to matrix algorithms using inaccurate matrices. However it has no effect when applied to an algorithm that already gives good accuracy. Since it is easy to accurately construct the derivative matrix for the matrix algorithm, the preconditioning has no value in computing the Chebyshev spectral derivative. However, in a situation where it was more difficult to accurately construct the matrix, it might be very useful. An example of this might be polynomial interpolation at the roots of a Legendre polynomial, or polynomial interpolation at some set of points which don't have any special structure.

The cosine transform-recursion algorithm (CFT) was less accurate than the FFT-recursion algorithm. Since the recursion part of these algorithms is the same, it is clear that the blame for this difference belongs to the cosine transform subroutine FCR. By looking at the figures 3 and 4 where the matrices use the trigonometric identity form but no flipping, we can see that the slope of the curve for the cosine transform algorithm is about the same as that for the matrix multiply algorithm. Since the cosine transform certainly involves calculating trigonometric functions, a reasonable hypothesis for the increased error is that those functions were being calculated inaccurately for the $\sin(\pi - x)$ reason discussed in the last section.

When the matrices used in the matrix algorithms were computed accurately, all of the algorithms except the cosine transform were able to achieve accuracy as good as the estimated minimum. This has two important consequences. One, when doing a computation that involves computing the Chebyshev derivative, considerations of accuracy should not influence the choice of algorithm. That choice should be made depending on other considerations, for example speed. Two, future research into reducing the error of the Chebyshev derivative should concentrate on changing the Chebyshev derivative operator itself, not on better algorithms for computing it. Better algorithms can't help.

7 CPU Timing for Various Algorithms

Now we will consider the other major consideration in choosing an algorithm for computing the Chebyshev derivative, CPU time. If the roundoff error of the matrix algorithms was $O(N^4\epsilon)$, we would avoid using them even if they were faster than the transform-recursion algorithm. We would

choose the transform-recursion algorithm regardless of its speed. But since we have shown that the roundoff error of the matrix algorithms can be at least as small as that of the transform-recursion algorithm, we need to know which algorithm is fastest. Unfortunately, there is no easy way to know which algorithm is fastest except to try them all out. Simply counting the number of operations is not a very reliable guide, especially on a vector or parallel computer.

So in this section we try them all out. We have tried to simulate the computations in a large spectral solution of a PDE as realistically as we could. We chose to use large vector computers on which to run tests, because those tend to be the ones on which large computations are run. We chose two-dimensional problems, i.e., simultaneously computing the derivative of many vectors, because the largest computations tend to be multidimensional. This has important implications for speed. A single FFT is quite hard to vectorize, and so the transform-recursion algorithm is at a disadvantage on a one-dimensional problem on a vector machine. The matrix algorithms tend to vectorize better. However a multiple FFT offers much more opportunities for vectorization, and conclusions drawn from one-dimensional tests might be inappropriate when applied to a two- or three-dimensional problem.

We computed the Chebyshev derivative on two different machines, using several different versions of the three algorithms, and compared the execution time. The first machine is the IBM 9121-320 VF. It has a 15 nanosecond cycle time, eight 256-element vector registers, and a 11-MFLOP scalar unit. Its vector unit can reach 133 MFLOPs on multiply-add operations and 66 MFLOPs for other operations. The second machine is a Cray 2. It is about 4 or 5 times faster than the IBM. Its cycle time is 4.1 nanoseconds. It has eight 64-element vector registers. It has 4 processors, but all of our computations were run on a single processor.

The two machines are not very different in their general architecture. Major differences are that the IBM has a 256-element vector registers, while the Cray has a 64-element vector registers. The Cray has gather-scatter hardware, while the IBM does not. The FORTRAN compiler on the Cray is probably slightly smarter than the one on the IBM. This should not play a big role since most of the computation was done with library subroutines. The Cray compiler was CFT77 version 4. The FORTRAN compiler on the IBM was VS FORTRAN Version 2. The libraries used on the Cray were SCILIB, and the cosine transform subroutine FCR was from LARCLIB. The library on the IBM was ESSL (Engineering and Scientific Software Library) version 4. The matrix multiplication subroutine used there was DGEMUL, and the multiple FFT was DCRFT. On the Cray the matrix multiplication subroutines MXM and MXMA were used and the multiple FFT was RFFTMLT.

We did our tests on two different machines to see how different the results would be. The results were quite large, compared to the differences between the architecture of the machines. Apparently, small differences in the machines make a big difference in which algorithm is fastest. This is why we have described the hardware and software of the machines in such detail. If a reader

wanted to verify our results, but did not use a setup that was almost exactly the same as ours, he might get timing results quite different from this paper. Unfortunately this also means that our results have little meaning to people who are not using exactly the same machine as ours.

7.1 Techniques in computing Chebyshev derivative

Each of the matrix algorithms can be computed in several ways. The matrix multiplications can be performed either by explicitly coded DO loops or by a library subroutine. A library subroutine would generally be faster. If explicit loops are used, what code results will depend on the compiler. On the Cray, apparently explicit loops that do a matrix multiplication are recognized and replaced by library subroutines, so there is no point in writing explicit loops. On the IBM, it is possible to use compiler directives to specify along which loop index to vectorize. Let N be the number of grid points, and M be the number of vectors being differentiated. If $N > M$, then it will probably be more efficient to vectorize in the N-direction, and if $N < M$, then it would probably be more efficient to vectorize in the M-direction. We tested both possibilities.

When using a library subroutine, something analogous is possible. In ESSL, the matrix multiplication subroutine has options for transposing either of the two operand matrices before multiplying. On the Cray, MXMA has these options and MXM does not. We are computing DA, where D is the derivative matrix, and A is the matrix whose columns are the vectors to be differentiated. By reversing the order of the operands, and specifying the transpose of both matrices, this equivalent to compute the transpose of DA. This would correspond to vectorizing in a different direction than the original subroutine call. If N is much different than M, this might be substantially faster.

7.2 Timing results on the IBM 9121-320 VF

On the IBM, the derivative was computed 9 different ways, for a variety of different values of N and M. They are the transform-recursion algorithm and 8 different ways using matrix algorithms: matrix multiply or even-odd algorithm, explicit loops or library subroutine, vectorization in the N-direction or the M-direction. Each computation was timed.

Figure 10 shows which algorithm was fastest for each value of N and M. With a few exceptions, the values where each algorithm is fastest fall into fairly distinct regions. If M is bigger than about 16, and N is bigger than about 300, the transform-recursion algorithm is the fastest. If M is 16 or less, then the even-odd algorithm is fastest for N less than about 450. If N was smaller than about 300 then the even-odd algorithm, using ESSL subroutines for the matrix multiplies was fastest for any value of M. If $N > M$ then vectorization in the N-direction was faster, otherwise vectorization in the M-direction was faster. The matrix multiply algorithm was never fastest, for any N or M. The explicit loops were never fastest.

Figure 11 show the worst algorithm for each choice of N and M. We see that for N less than

about 140, the transform-recursion algorithm is the slowest, even slower than the matrix multiply with explicit loops. For N bigger than 140, the matrix multiply with explicit loops was always slowest.

Figures 12 and 13 show CPU time (in microseconds) vs. M for the case $N = 64$. The data have been normalized by M, so any deviation from a flat straight line reflects differences in the degree of vectorization. The graph is rather hard to interpret, but some observations are possible. When comparing vectorization in the N direction with vectorization in the M direction, the CPU time of the explicit loops varied much more than did the ESSL subroutines. In the even-odd algorithm, using the ESSL subroutines, there was almost no difference in CPU time between vectorizing the N and M directions, but with the matrix multiply algorithm and the ESSL subroutines, there were differences in CPU time of up to 10 or 20 percent between vectorizing in N and M.

Figure 14 shows CPU time vs. N for the case $N = M$. The CPU time has been normalized by N^3, so that the curves for the matrix multiply algorithms would be flat lines if the computations were all computed at the same speed. It is even harder to interpret than the previous results, but we will try. We would expect no difference between vectorizing in the N and M directions. When ESSL subroutines are used, there is little difference. But when explicit loops are used there are substantial differences. Another observation is that if only the ESSL subroutines are considered, the even-odd algorithm is faster than the matrix multiply by a factor of about $1.6 - 1.8$, and this is true for all values of N except possibly for the smallest ones.

7.3 Timing results on the Cray-2

The preceding results were all for the IBM. There were some differences in the computations we did on the Cray. We did not use explicit DO loops. We computed the matrix algorithms with vectorization in the N direction and with vectorization in the M-direction. This was accomplished with the computing-the-transpose trick described earlier. On the Cray, there is a cosine transform (CFT) subroutine available from LARCLIB. On the IBM, there is a cosine transform subroutine in ESSL, but only in single precision, which does not meet our specification.

Figure 15 shows the fastest algorithm on the Cray for different values of N and M. It is quite different from the same graph for the IBM. Unlike the IBM, there is a region where the matrix multiply algorithm is fastest. If N is less than about 20, the matrix multiply is fastest for all M. If N is less than about 50 and M is less than about 175, it is also fastest. The region where the even-odd algorithm is fastest is smaller than it was on the IBM. For large M, the even-odd algorithm is fastest if N is between about 20 and about 100. For smaller M, it is fastest for somewhat larger values of N, and for M less than about 20, it is fastest for N as large as 400. Except for a few points where the FFT-recursion algorithm is fastest, all the other points belong to the CFT-recursion. Figure 16 shows the slowest algorithm on the Cray.

While the cosine transform is very fast, it has considerably more roundoff error than the other algorithms, especially for large N. For this reason we have duplicated timing results in figure 15 without including the timing of the CFT-recursion (figure 17). The region belonging to the matrix multiply is unchanged. Most of the region that belonged to the CFT-recursion was claimed by the FFT-recursion. The most interesting difference is that now the even-odd algorithm is fastest for all N less than about 200 (except for the matrix multiply region).

Figure 18 shows CPU time in microseconds, normalized by M for the case $N = 64$. In this case, there are differences on the order of $25 - 50\%$ in CPU time between vectorizing in N and M.

Figure 19 shows CPU time in microseconds, normalized by N^3 for the case $N = M$. We note that except for one case, $N = M = 150$, there is little difference between vectorizing in N or M.

8 Conclusions about CPU timing

On the IBM, the even-odd algorithm is almost always the fastest algorithm. It would be rare that a calculation would take so many collocation points that the FFT-recursion would be faster. However the situation is different on the Cray-2. The even-odd algorithm is not as successful, due partly to the existence of the fast cosine transform subroutines in two-dimensional problems. Though fast, highly-optimized cosine transform subroutines are harder to obtain than highly-optimized FFT subroutines. This is one reason to look at the relative speed of the algorithms without considering the cosine transform. Another is the fact that the cosine transform used in this study have much larger roundoff error than other algorithms.

For small values of M, such as would be encountered in a one-dimensional calculation, the even-odd algorithm is very successful on both the IBM and the Cray.

There are two important conclusions to be drawn. The first is that which algorithm is fastest depends on small details of the hardware and software. Probably experiments on another machine than these two would give completely different results than we have reported here. Probably the only reliable way to determine which algorithm to use is to test them all out on your machine.

The other conclusion is that two- or three-dimensions are different than one-dimension, because of the difference in ease of vectorization.

Acknowledgments

Both authors would like to thank Dr. Ernest Rothman for many helpful discussion and Mr. George Loriot for technical information about computer software and hardware. This research was supported by

References

Canuto, C., Quarteroni, A., Hussaini, M.Y., and Zang, T. (1988). *Spectral Methods in Fluid Mechanics*, Springer-Verlag, New York.

Breuer, K., and Everson, R. (1989). On the Errors Incurred Calculating Derivatives Using Chebyshev Polynomials, submitted to *J. Comput. Phys.*. Report # 89-190, Center for Fluid Mechanics, Turbulence, and Computation, Brown University.

Gottlieb, D., and Orszag, S. (1977). Numerical Analysis of Spectral Methods: Theory and Applications, SIAM, Philadelphia.

Rothman E. Reducing Round-off Error in Chebyshev Pseudospectral Computations. In Durand, M., El Dabaghi, F. (eds.), paper presented at *The 2nd Symposium on High-Performance Computing*, Montpellier, France, 7-9 October 1991.

Solomonoff A. (1992). A Fast Algorithm For Spectral Differentiation, *J. Comput. Phys.*, to be published.

Trefethen, L.N., and Trummer, M.R. (1987). An Instability Phenomenon in Spectral Methods, *SIAM J. Numer. Anal.*, **24**, 1008-1023.

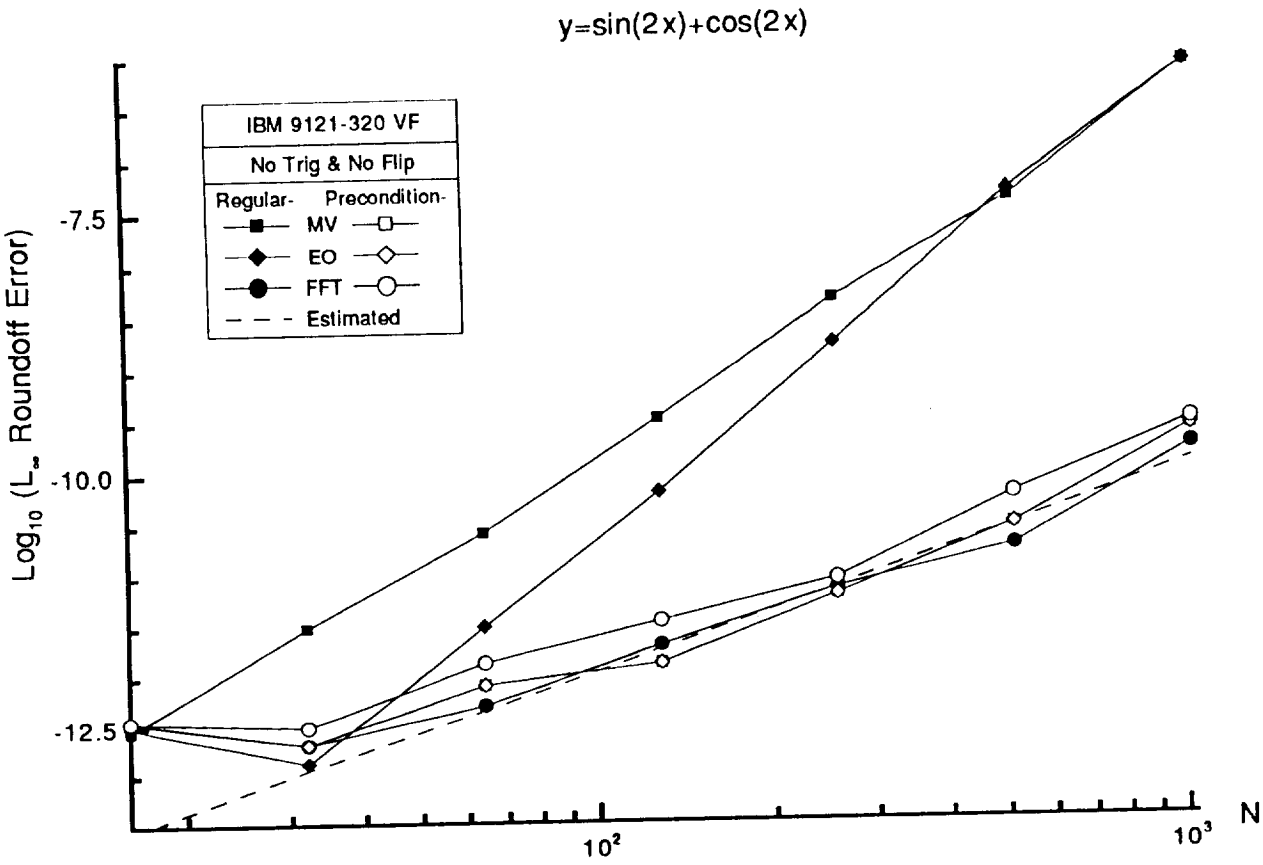


Figure 1: L_∞ roundoff error of $u(x) = \sin(2x) + \cos(2x)$ without using both trigonometric identities and flipping on IBM 9121-320 VF.

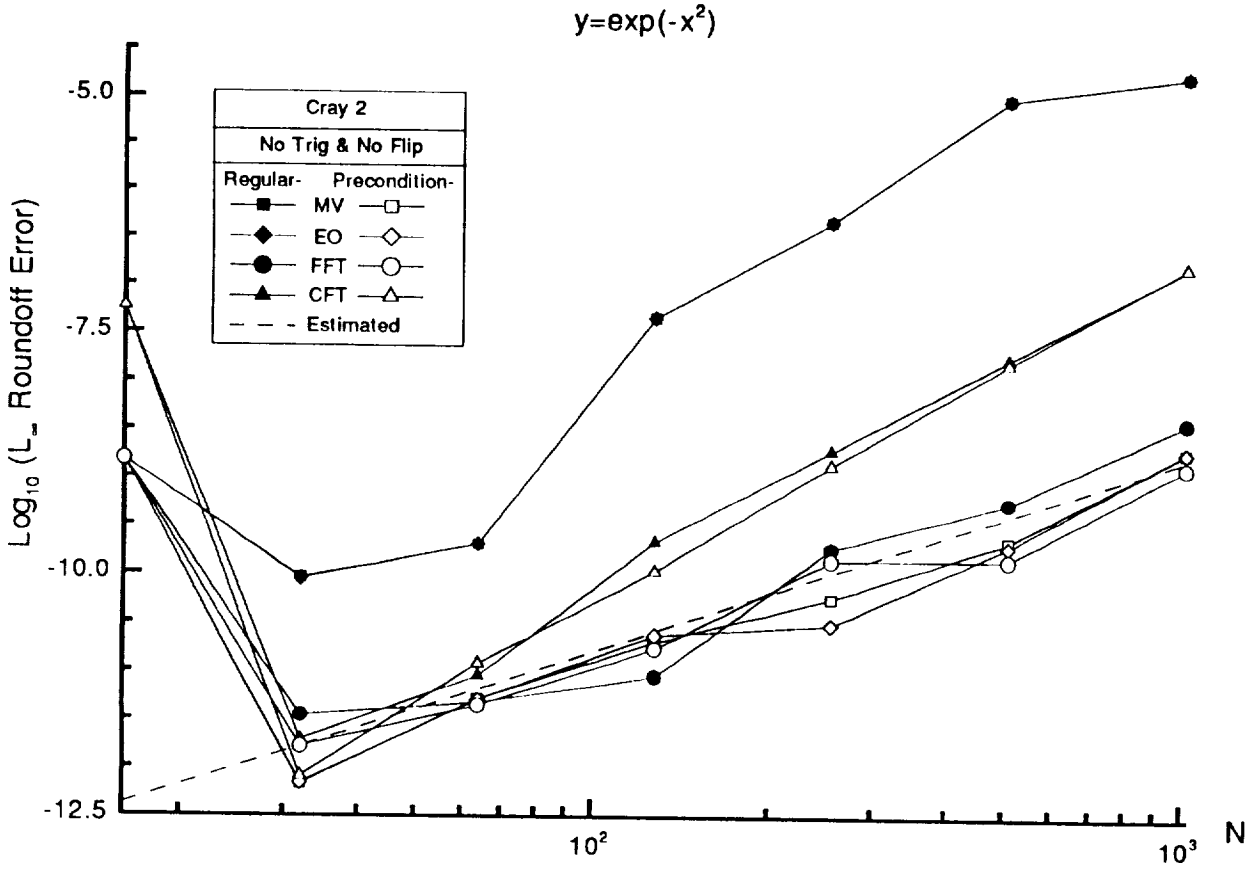


Figure 2: L_∞ roundoff error of $u(x) = \exp(-x^2)$ without using both trigonometric identities and flipping on Cray-2.

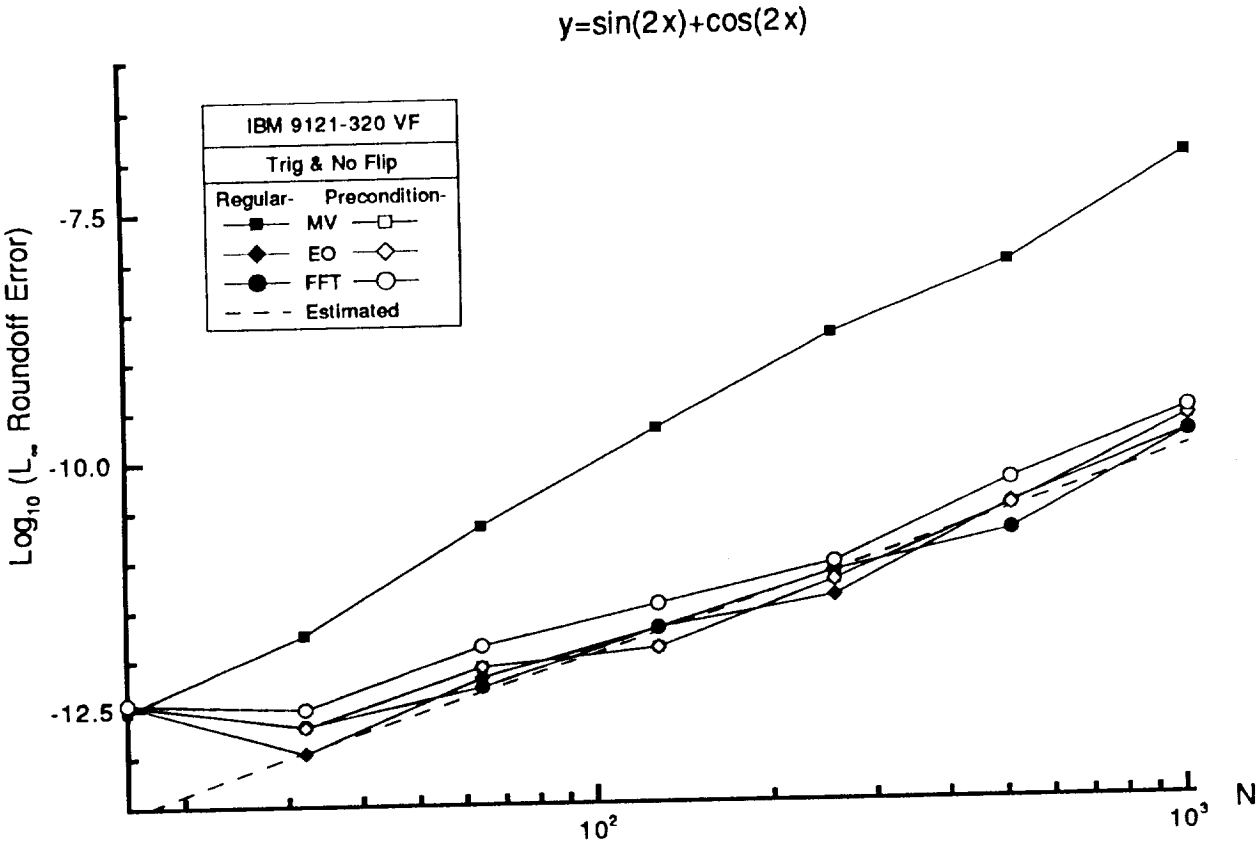


Figure 3: L_∞ roundoff error of $u(x) = \sin(2x) + \cos(2x)$ with trigonometric identities but no flipping on IBM 9121-320 VF.

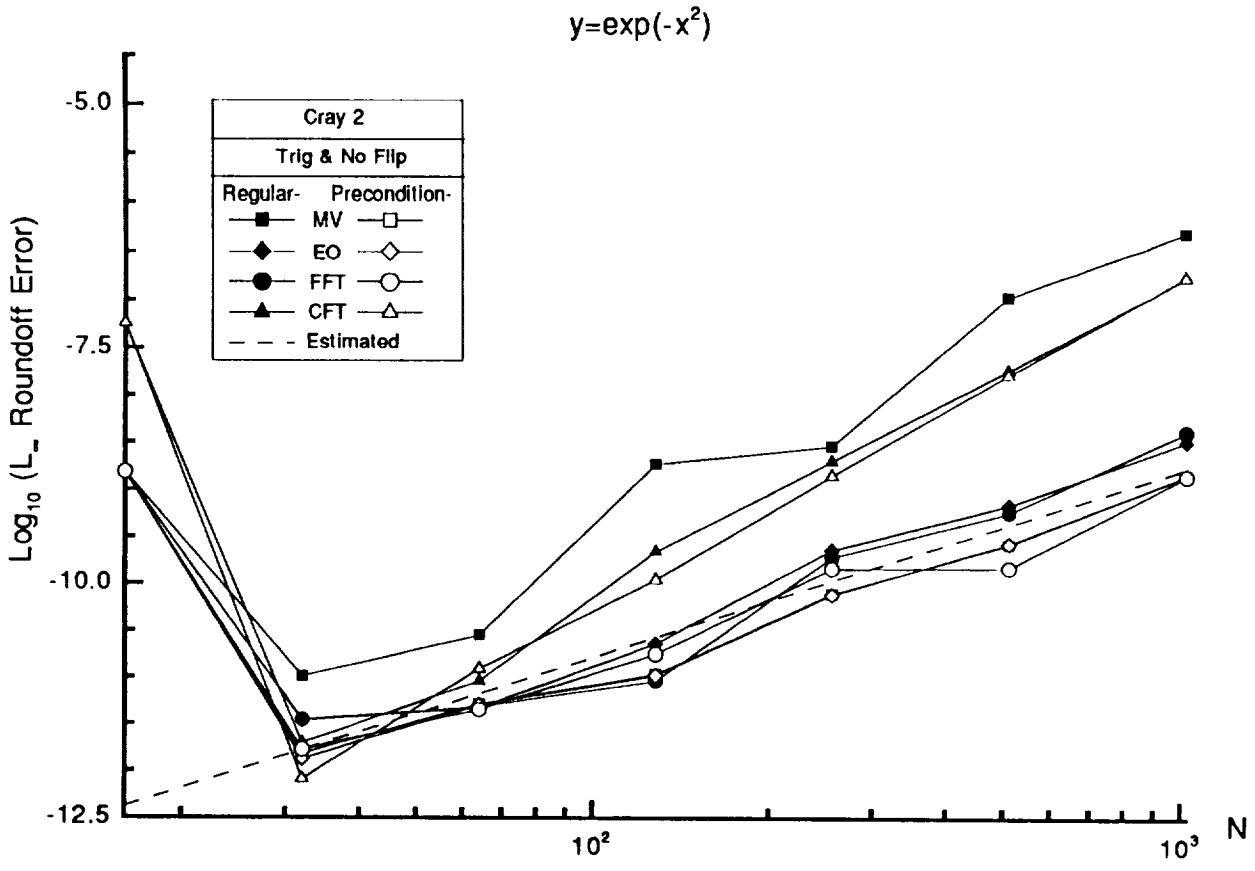


Figure 4: L_∞ roundoff error of $u(x) = \exp(-x^2)$ with trigonometric identities but no flipping on Cray-2.

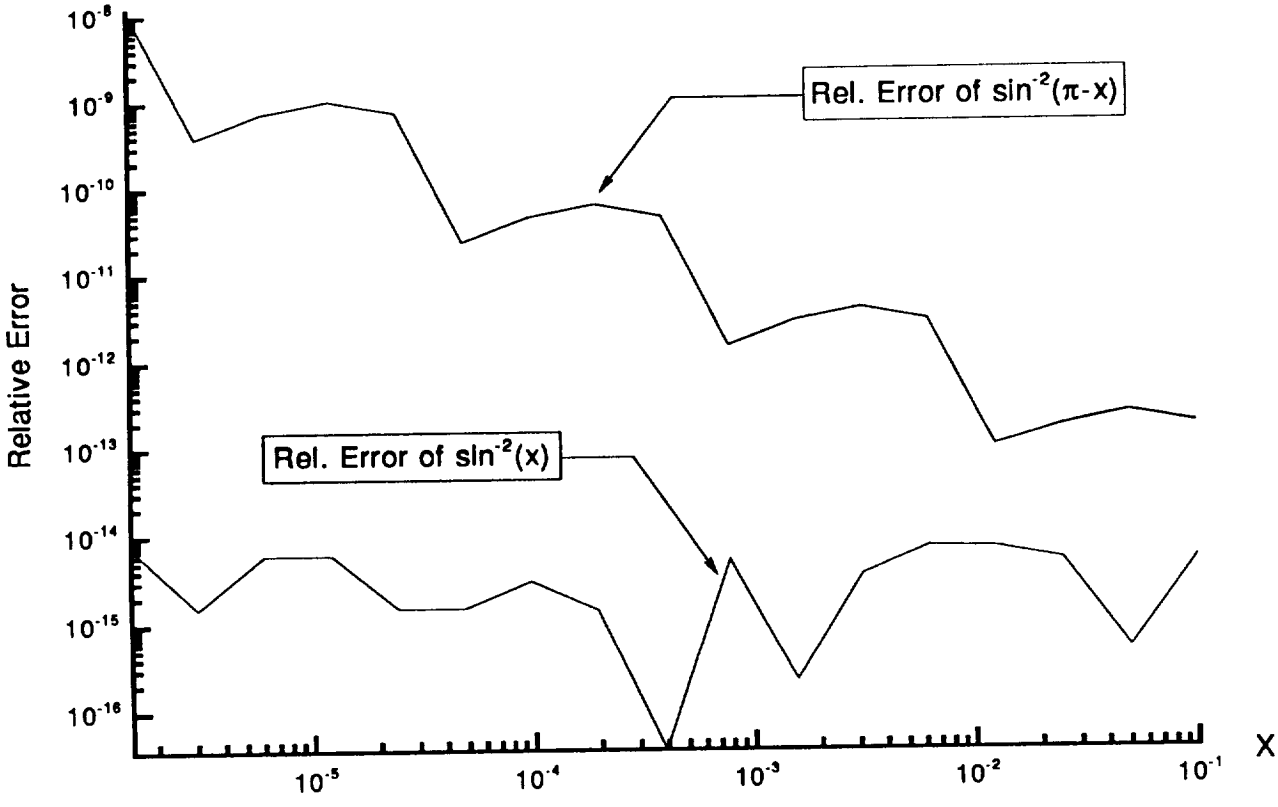


Figure 5: Relative error of $\sin^{-2}(x)$ and $\sin^{-2}(\pi - x)$

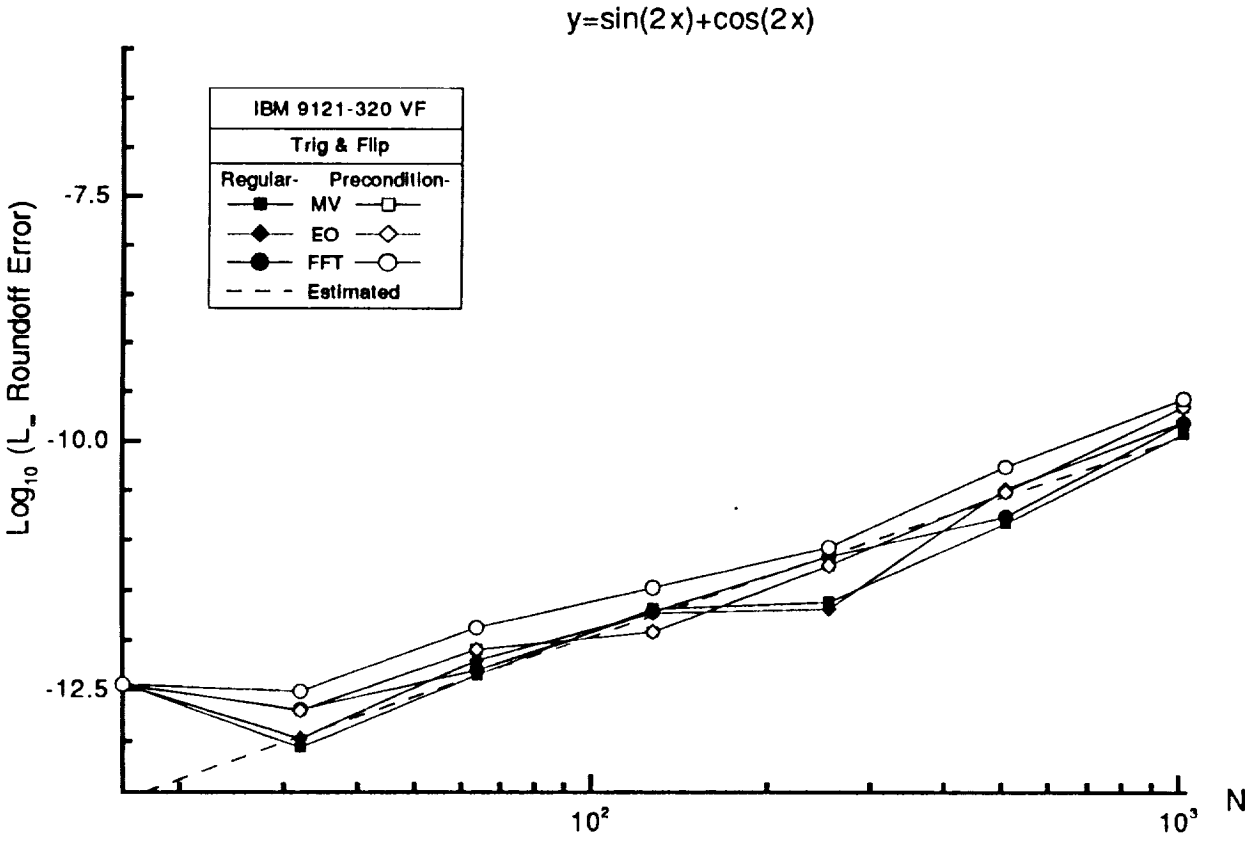


Figure 6: L_∞ roundoff error of $u(x) = \sin(2x) + \cos(2x)$ using both trigonometric identities and flipping on IBM 9121-320 VF.

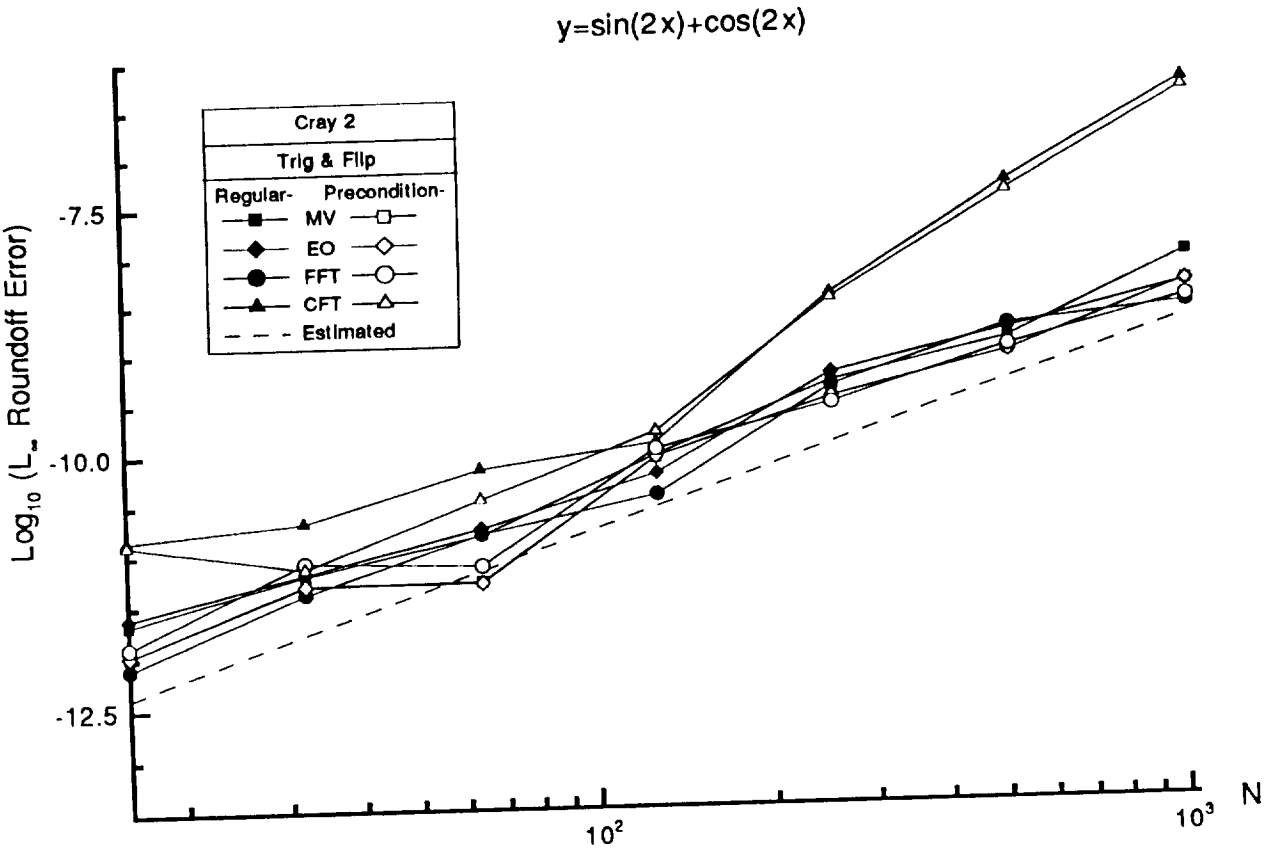


Figure 7: L_∞ roundoff error of $u(x) = \sin(2x) + \cos(2x)$ using both trigonometric identities and flipping on Cray-2.

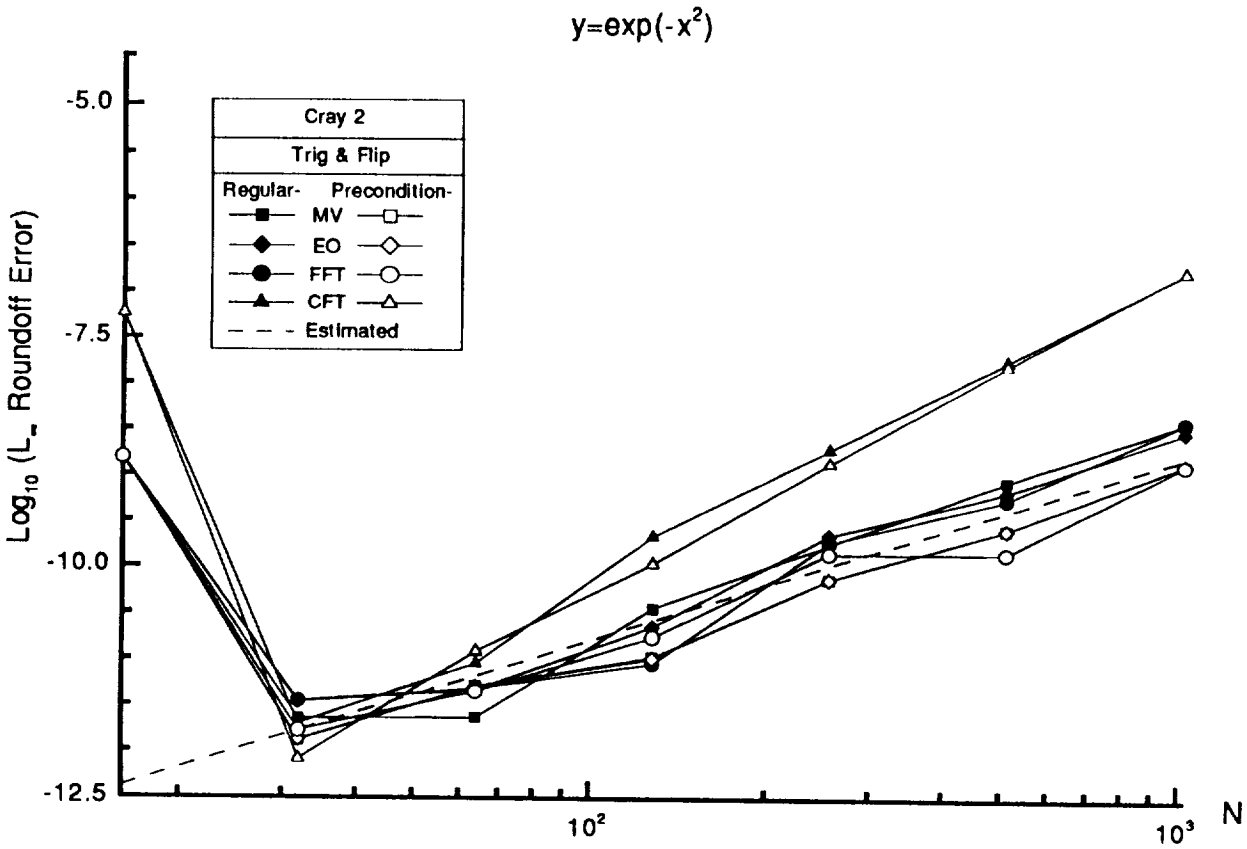


Figure 8: L_∞ roundoff error of $u(x) = \exp(-x^2)$ using both trigonometric identities and flipping on Cray-2.

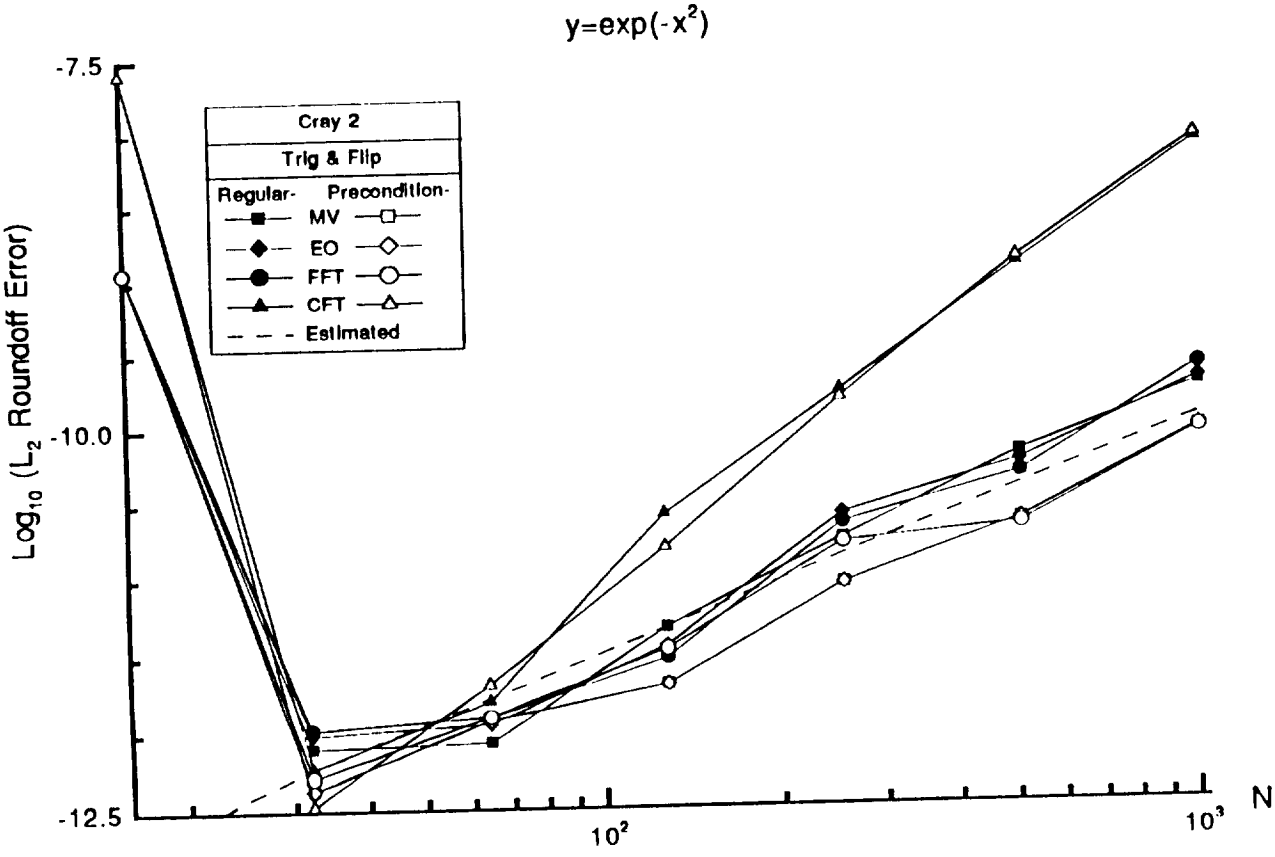


Figure 9: L_2 roundoff error of $u(x) = \exp(-x^2)$ using both trigonometric identities and flipping on Cray-2.

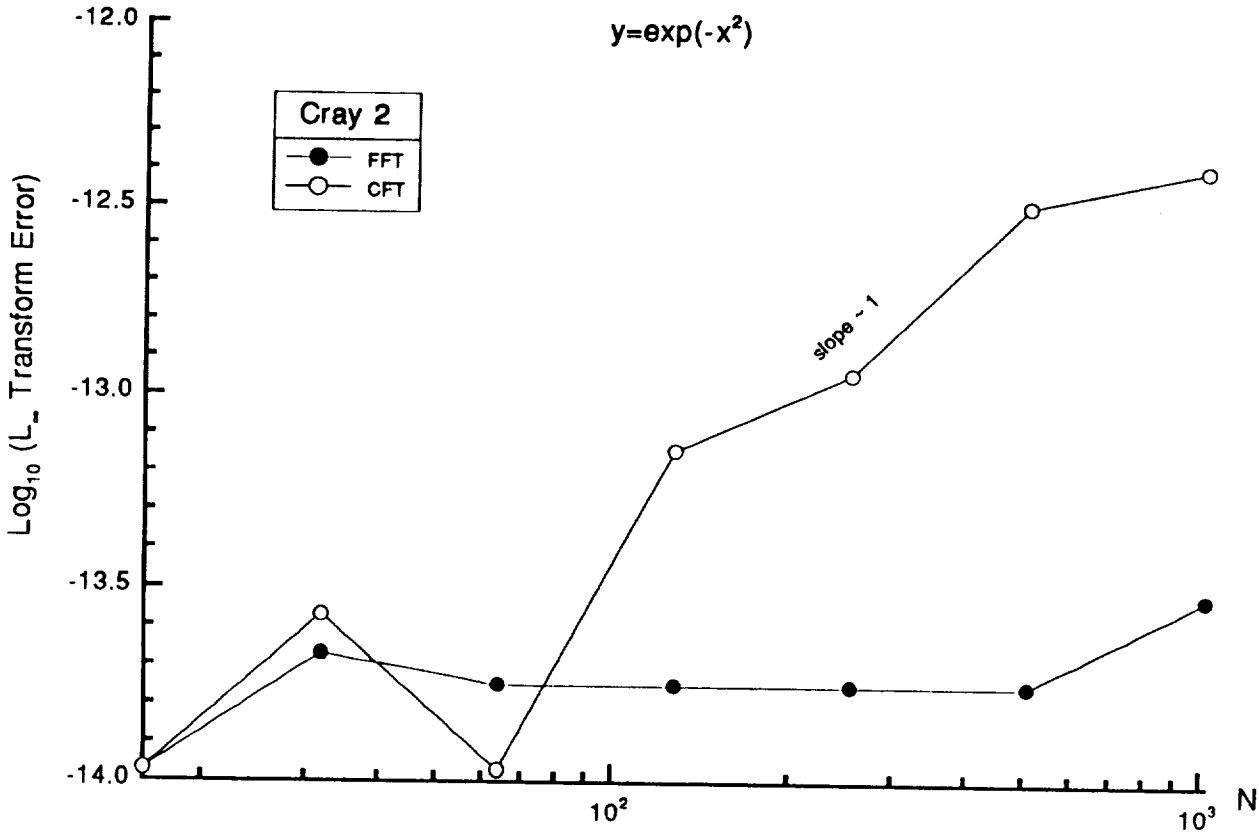


Figure 10: L_∞ roundoff error of the forward and backward transform of $u(x) = \exp(-x^2)$ using the cosine transform and fourier transform algorithms.

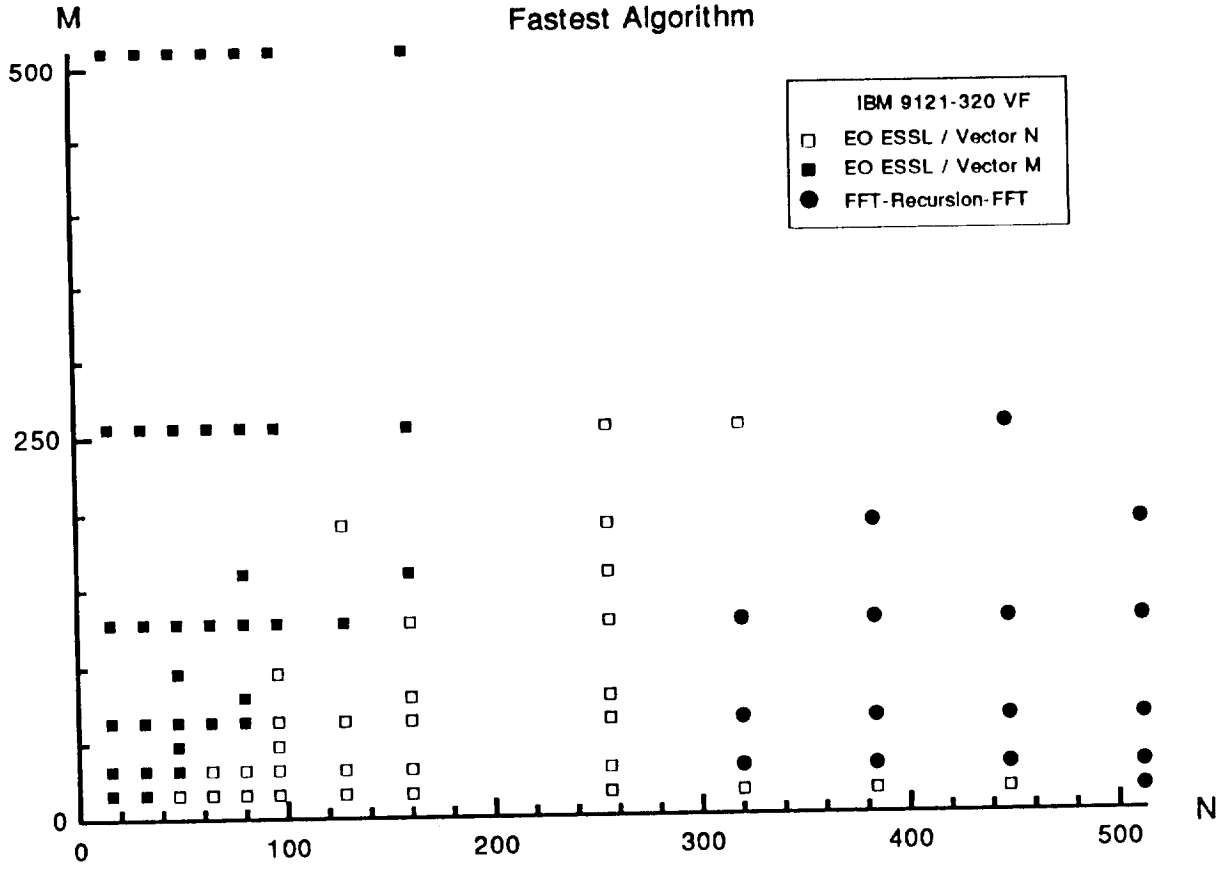


Figure 11: Fastest algorithms on IBM 9121-320 VF

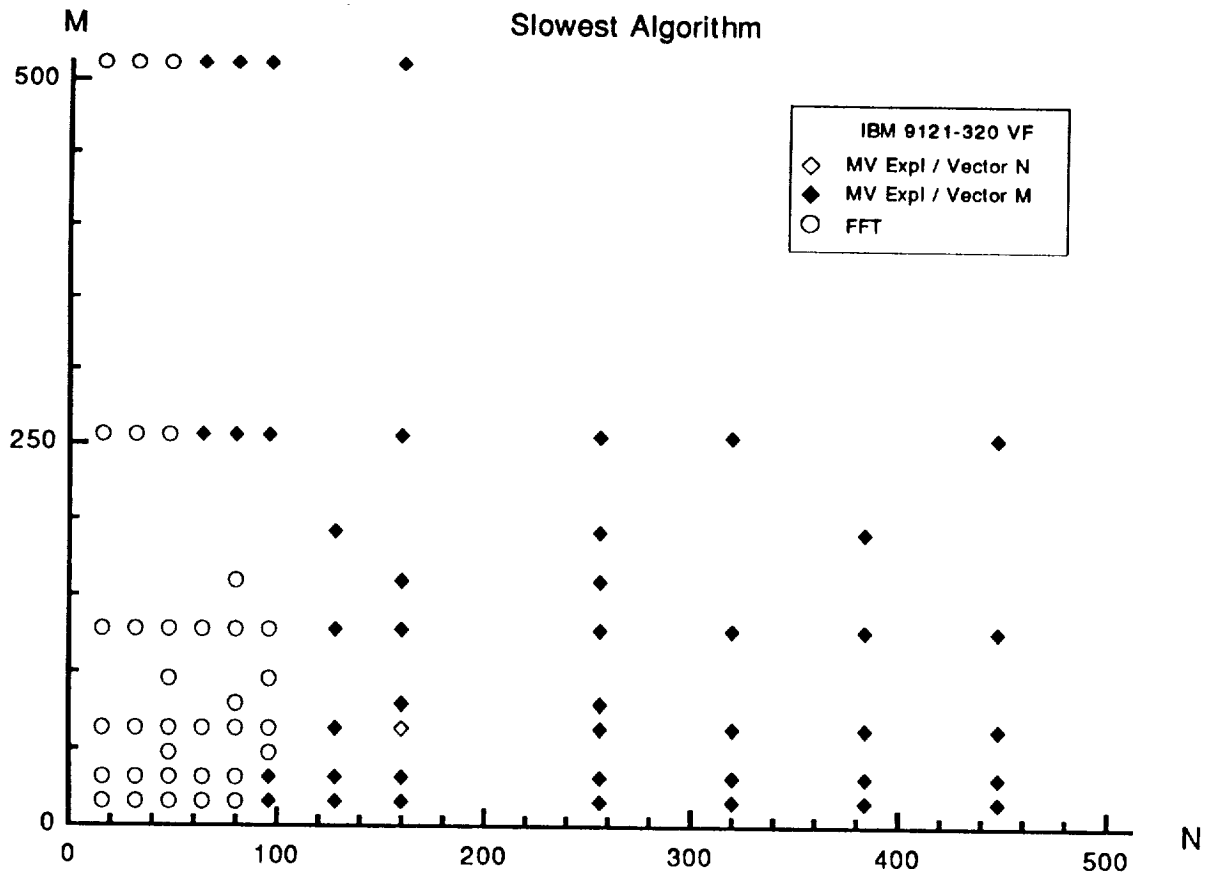


Figure 12: Slowest algorithms on IBM 9121-320 VF

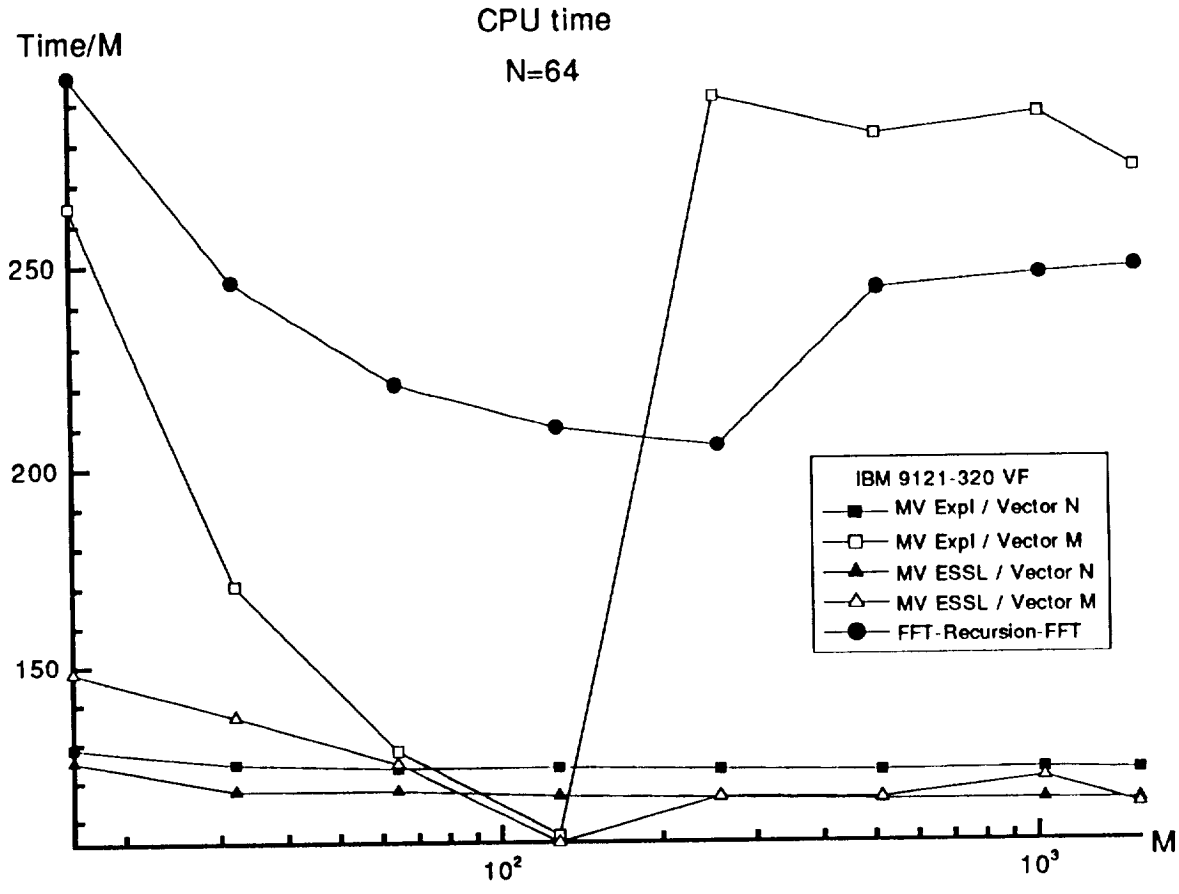


Figure 13: CPU timing of Matrix multiply and FFT-Recursion algorithms vs. M, the number of vector being differentiated, for fixed $N = 64$ on IBM 9121-320 VF

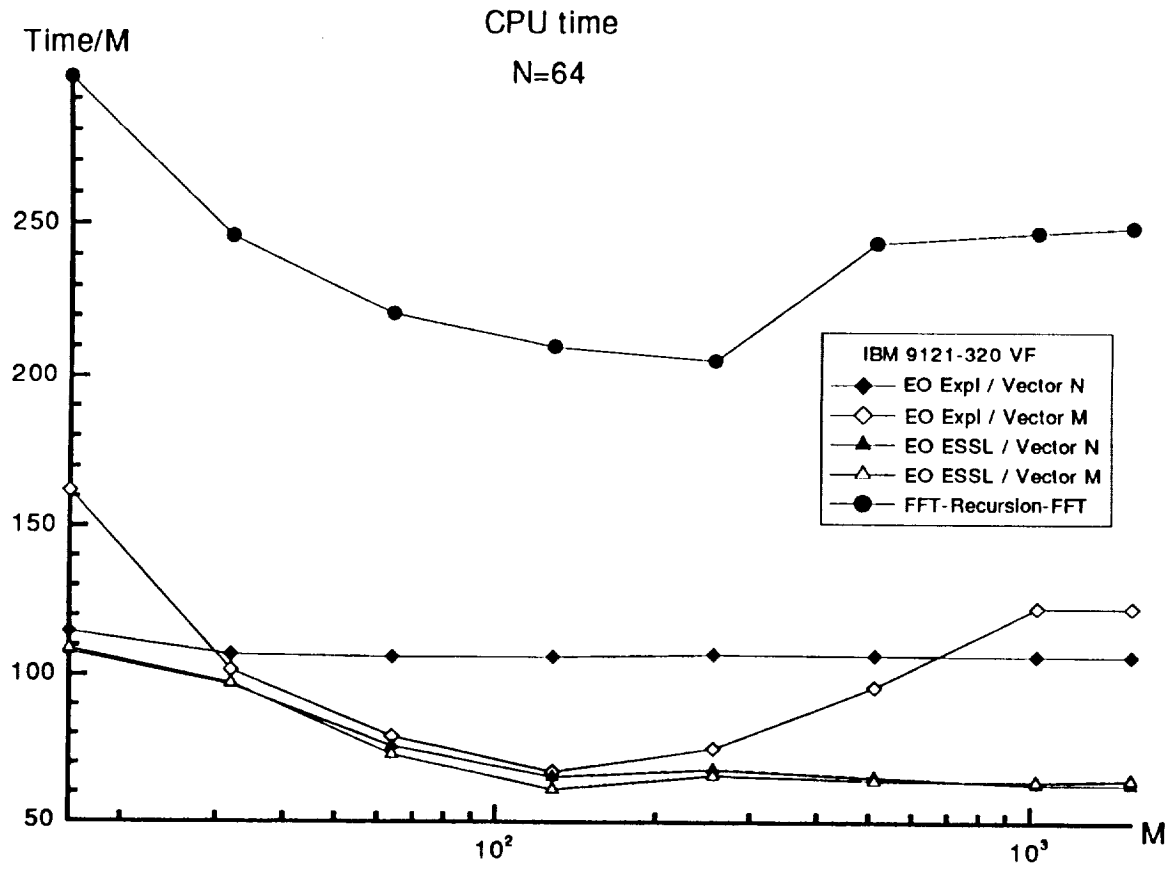


Figure 14: CPU timing of Even-Odd and FFT-Recursion algorithms vs. M, the number of vector being differentiated, for fixed $N = 64$ on IBM 9121-320 VF

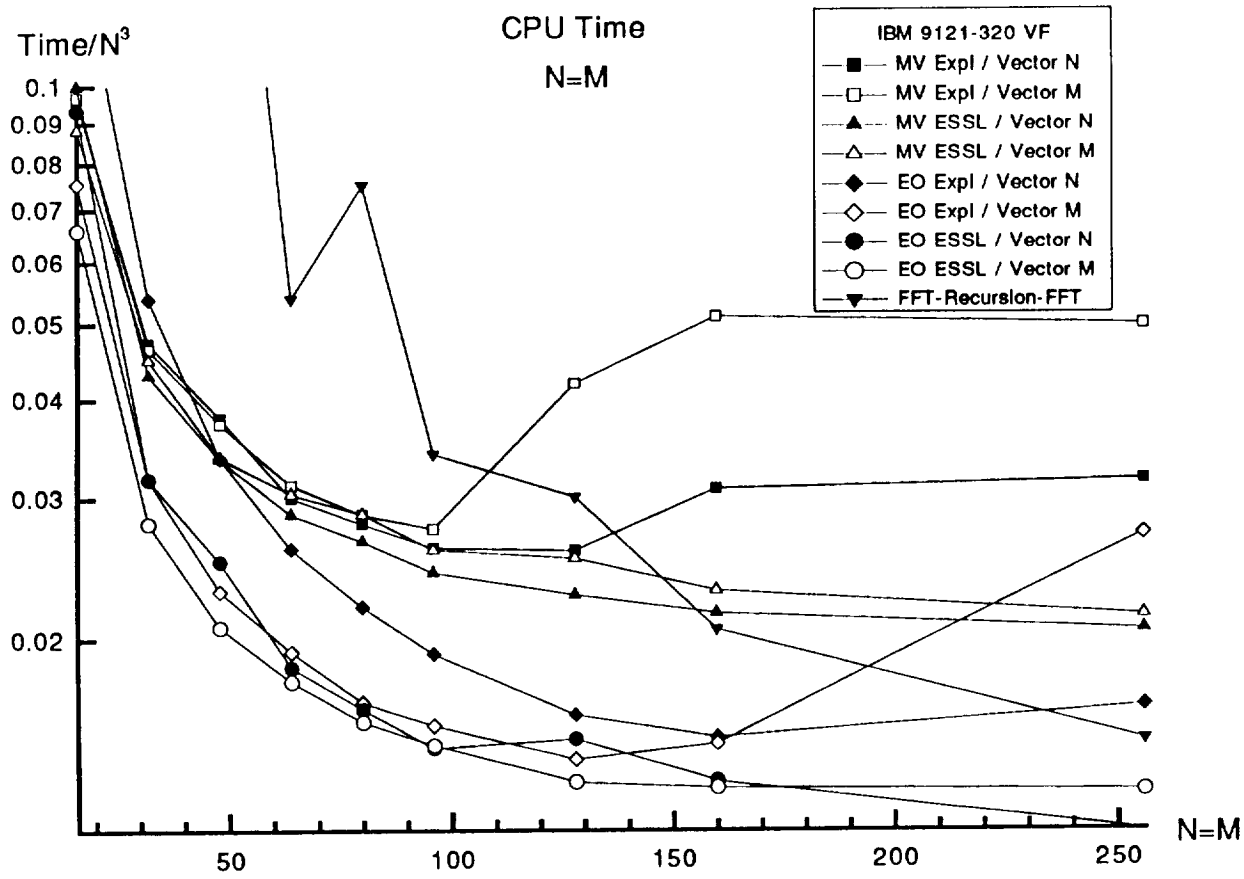


Figure 15: CPU timing of all differentiation algorithms vs. $N = M$ on IBM 9121-320 VF

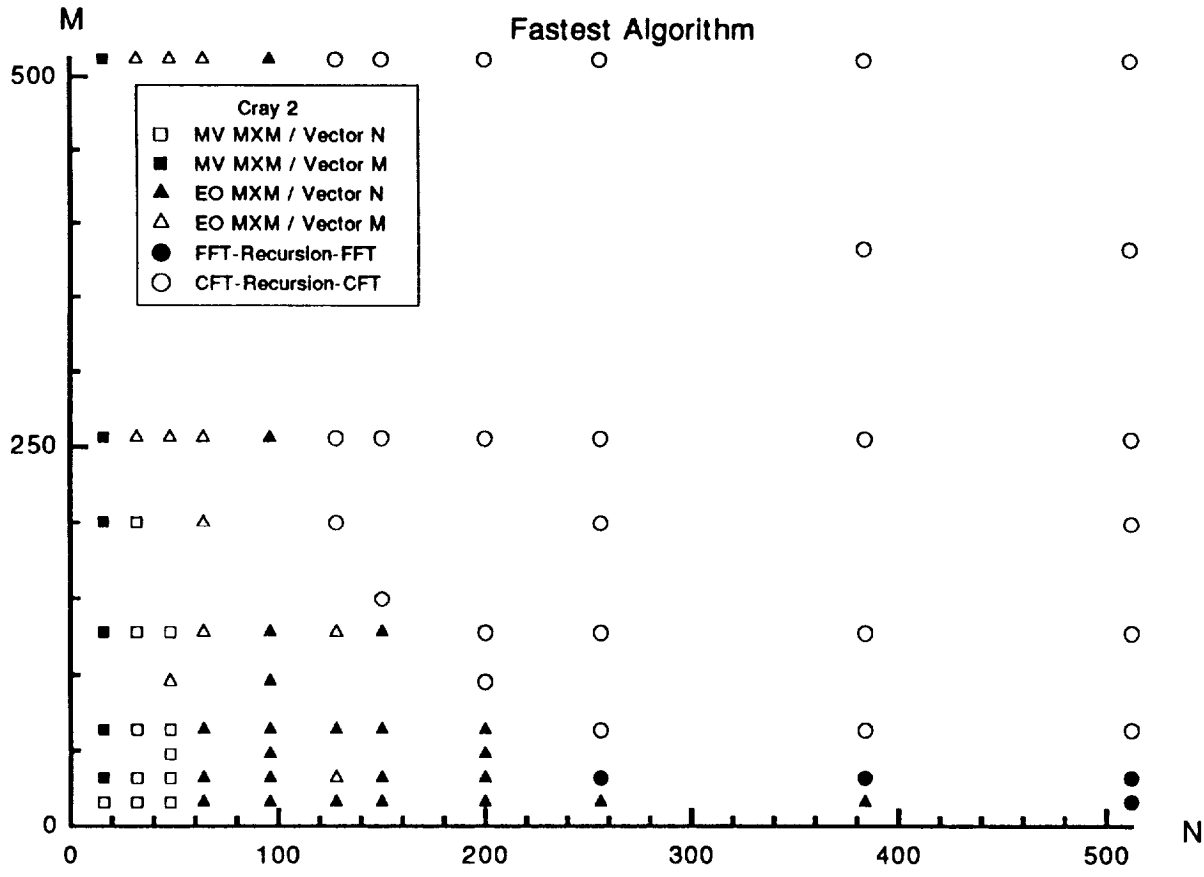


Figure 16: Fastest algorithms on Cray 2

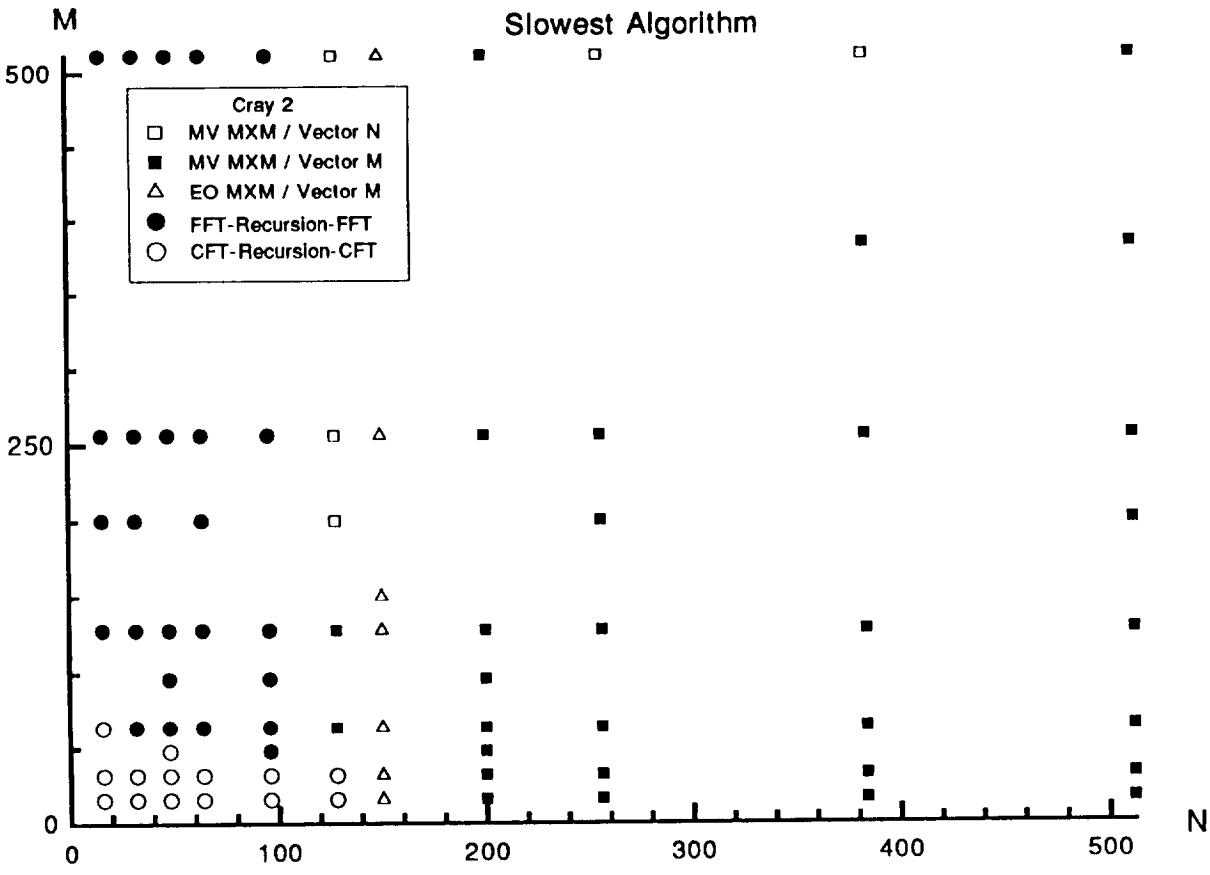


Figure 17: Slowest algorithms on Cray 2

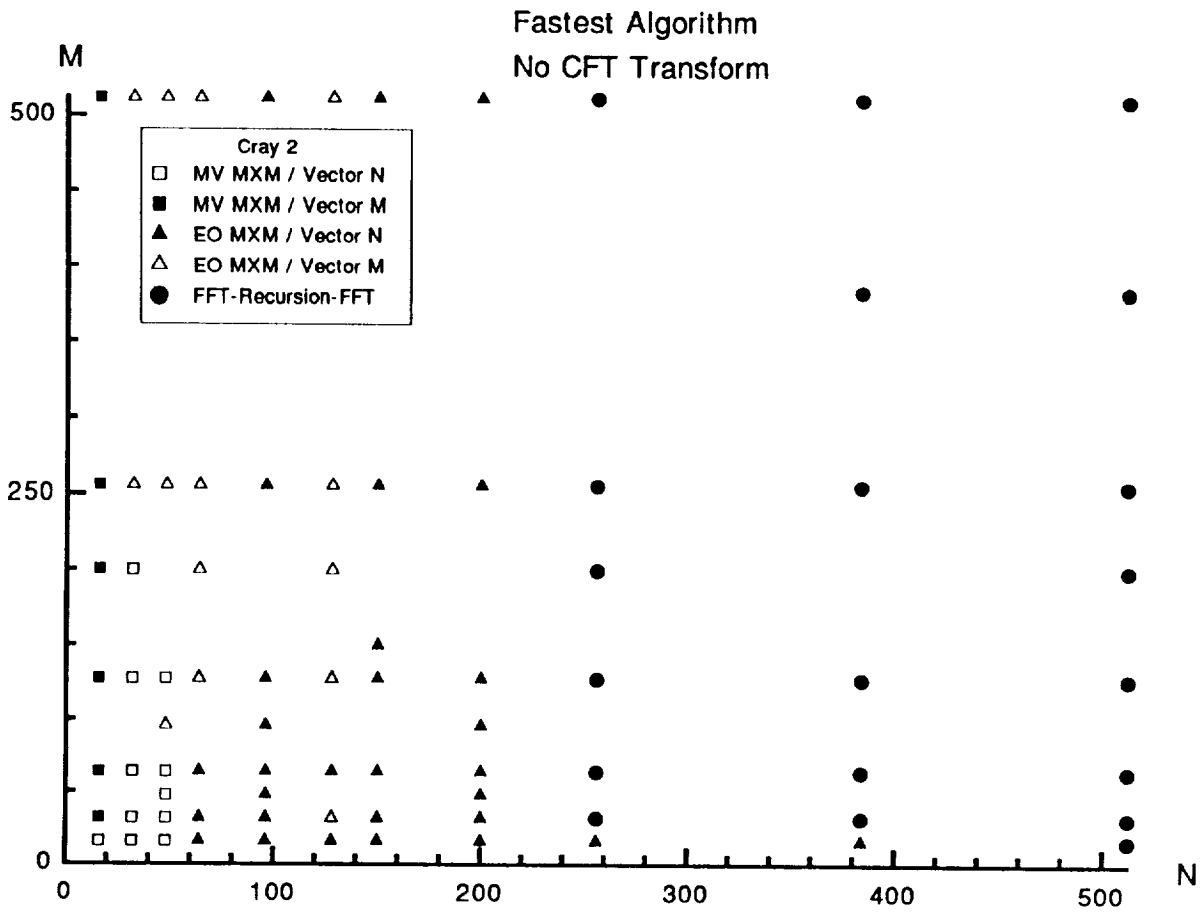


Figure 18: Fastest algorithms (excluding the CFT-recursion algorithm) on Cray 2

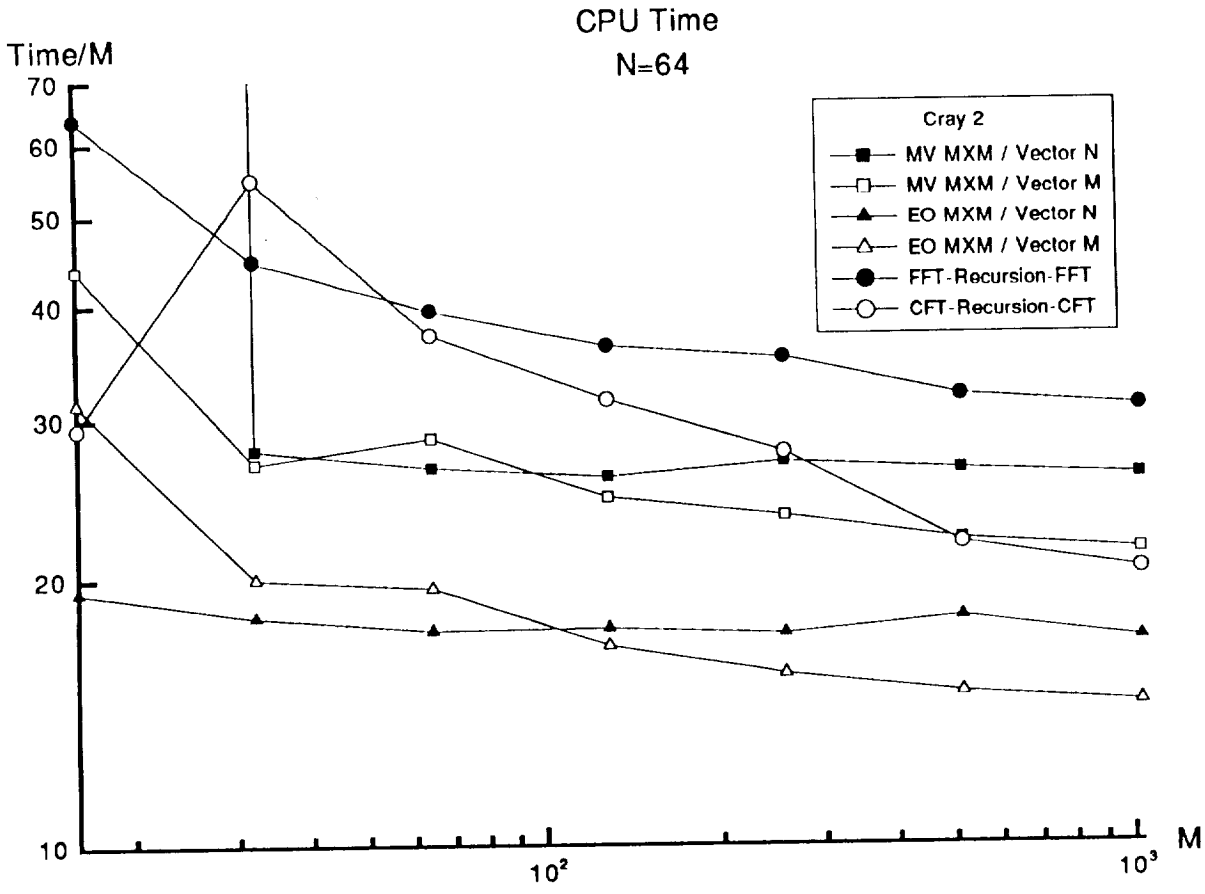


Figure 19: CPU timing (μs) of all differentiation algorithms vs. M, the number of vector being differentiated, for fixed $N = 64$ on Cray 2

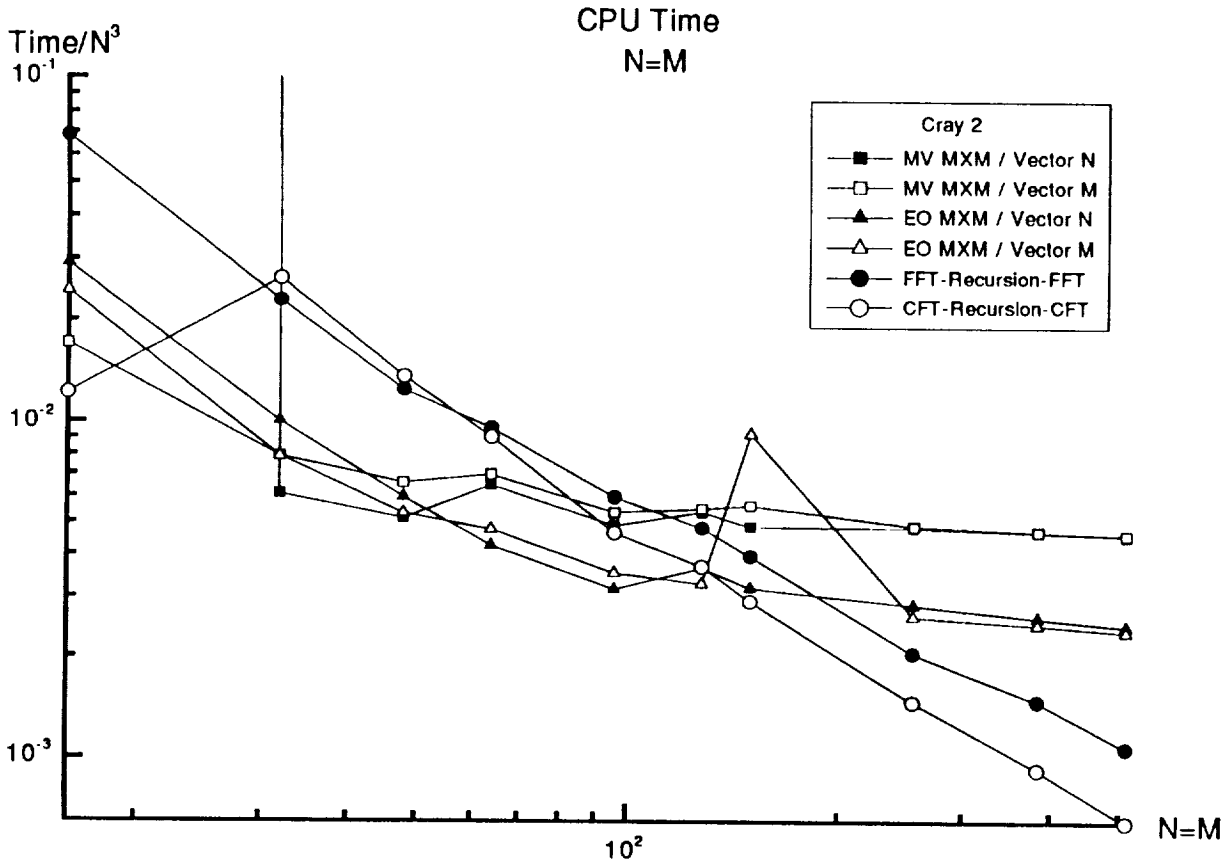


Figure 20: CPU timing (μs) of all differentiation algorithms vs. $N = M$ on Cray 2

NASA
National Aeronautics and Space Administration

Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA CR-4411		

4. Title and Subtitle
Accuracy and Speed in Computing the Chebyshev Collocation Derivative

5. Report Date
December 1991

6. Performing Organization Code

7. Author(s)
Wai Sun Don
Alex Solomonoff

8. Performing Organization Report No.

9. Performing Organization Name and Address
Brown University
Division of Applied Mathematics
Providence, RI 02912

10. Work Unit No.
505-62-40-07

11. Contract or Grant No.
NAG1-1145

13. Type of Report and Period Covered
Contractor Report

12. Sponsoring Agency Name and Address
National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23665-5225

14. Sponsoring Agency Code

15. Supplementary Notes

Wai Sun Don: Division of Applied Mathematics, Brown University
Alex Solomonoff: Division of Applied Mathematics, Brown University

Langley Technical Monitor: Michele G. Macaraeg

16. Abstract

We study several algorithms for computing the Chebyshev spectral derivative and compare their roundoff error. For a large number of collocation points, the elements of the Chebyshev differentiation matrix, if constructed in the usual way, are not computed accurately. A subtle cause is found to account for the poor accuracy when computing the derivative by the matrix-vector multiplication method. Methods for accurately computing the elements of the matrix are presented, and we find that if the entries of the matrix are computed accurately, the roundoff error of the matrix-vector multiplication is as small as that of the transform-recursion algorithm.

Furthermore, results of the CPU time usage are shown for several different algorithms for computing the derivative by the Chebyshev collocation method for a wide variety of two-dimensional grid sizes on both an IBM and a Cray 2 computer. We find that which algorithm is fastest on a particular machine depends not only on the grid size, but also on small details of the computer hardware, as well. For most practical grid sizes used in computation, the even-odd decomposition algorithm is found to be faster than transform-recursion method.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
Chebyshev collocation Roundoff error CPU timing Matrix-Vector multiply Fast Fourier Transform	Unclassified-Unlimited Subject Category: 02

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of pages	22. Price
Unclassified	Unclassified	36	A03

NASA FORM 1626 OCT 86

For sale by the National Technical Information Service, Springfield, Virginia 22161-2171 NASA-Langley, 1991